

Building Neural Networks with the TensorFlow and Keras API

Welcome to the **Building Neural Networks with the TensorFlow and Keras API** module.

In this module, you learn to ...

01	Describe activation functions, loss, and optimizers
02	Build a DNN model using the Keras Sequential and Functional APIs
03	Use Keras preprocessing layers
04	Save/load and deploy a Keras model
05	Describe model subclassing

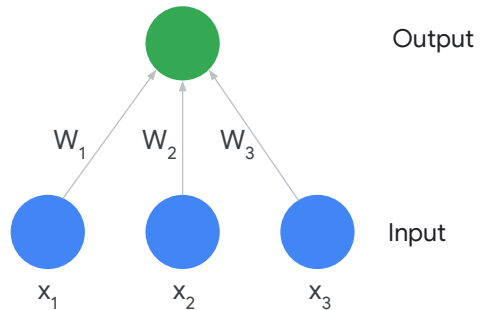


In this module, you'll learn to:

- Describe how activation functions, loss and optimizers work
- Build a DNN model using the Keras Sequential and Functional API's
- Use feature columns and Keras preprocessing layers in a Keras model
- Save/load, and deploy a Keras model on Vertex AI
- Describe model subclassing

A Linear Model can be represented as nodes and edges

$$\text{Output} = W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



Here is a graphical representation of a linear model.

We have three inputs on the bottom, x_1 , x_2 , and x_3 , shown by the blue circles. They're combined with some weight (w), given to them on each edge (those are the arrows pointing up), to produce an output (which is the green circle).

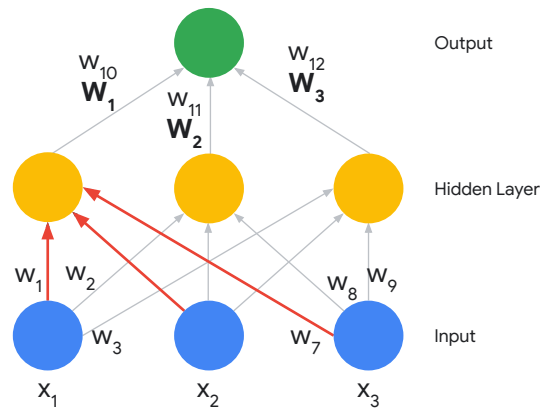
There often is an extra bias term, but for simplicity it isn't shown here. This is a linear model since it is of the form $y = w_1 * x_1 + w_2 * x_2 + w_3 * x_3$.

Add Complexity: Non-Linear?

$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$\begin{aligned} & ((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x_1 \\ & + ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x_2 \\ & + ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x_3 \end{aligned}$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



We can substitute each group of weights for a new weight. Look familiar?

This is exactly the same linear model as before despite adding a hidden layer of neurons.

So what happened?

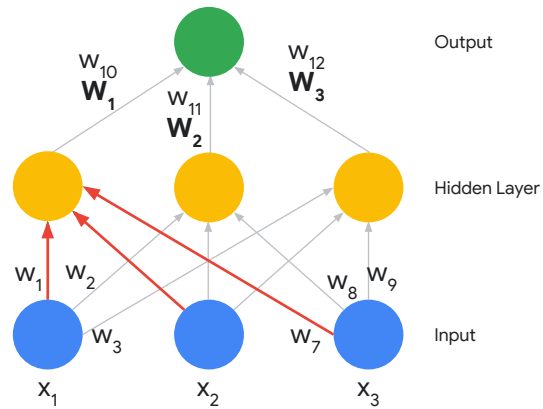
The first neuron of the Hidden Layer on the left takes the weights from all three Input nodes (those are the red arrows).

Add Complexity: Non-Linear?

$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$\begin{aligned} & ((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x_1 \\ & + ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x_2 \\ & + ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x_3 \end{aligned}$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



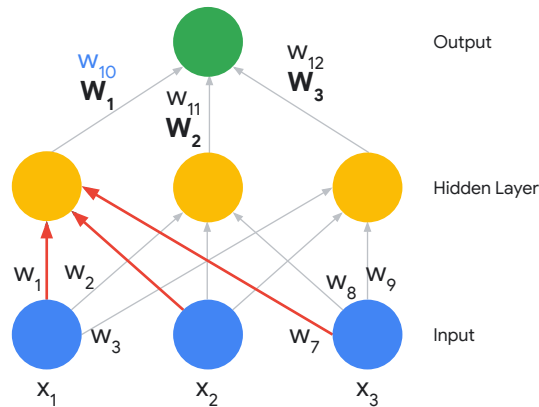
And those are little w_1 w_4 and w_7 respectively as you see highlighted here.

Add Complexity: Non-Linear?

$$\text{Output} = w_{10} * h_1 + w_{11} * h_2 + w_{12} * h_3$$

$$\begin{aligned} & ((w_{10} * w_1) + (w_{10} * w_4) + (w_{10} * w_7)) * x_1 \\ & + ((w_{11} * w_1) + (w_{11} * w_4) + (w_{11} * w_7)) * x_2 \\ & + ((w_{12} * w_1) + (w_{12} * w_4) + (w_{12} * w_7)) * x_3 \end{aligned}$$

$$= W_1 * x_1 + W_2 * x_2 + W_3 * x_3$$



And then you take the new weight that is the output of the first neuron which is our little w_{10} now and one of three weights into the final output.

You can see that we'll do this two more times for the other two yellow neurons and their inputs from x_1 x_2 and x_3 respectively.

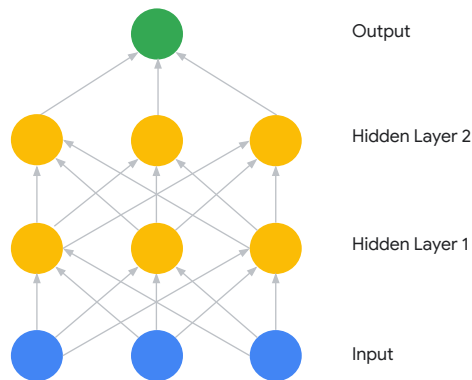
You can see that there is a ton of matrix multiplication going on behind the scenes. Honestly in my experience machine learning is basically taking arrays of various dimensionality and multiplying them against each other (where one array aka "Tensor" could be the randomized starting weights of the model and the other is the input dataset and yet a third is the output of a hidden layer).

You see it's all just simple math but a lot of it done really quickly.

Here still though we have a Linear model.

Let's go deeper!

Add Complexity: Non-Linear?

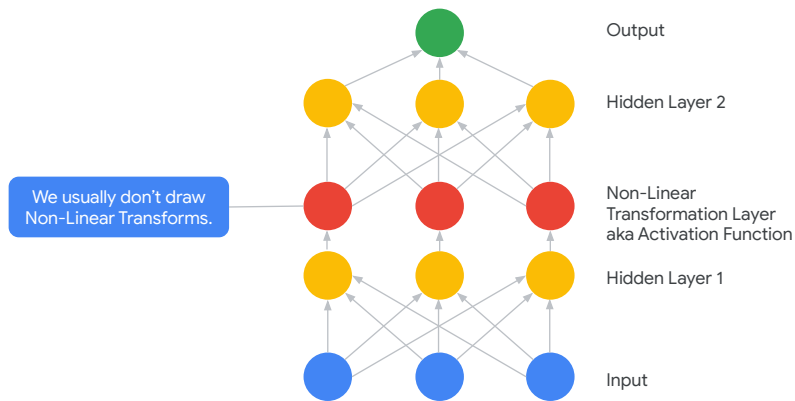


I know what you're thinking ... What if we added another hidden layer?

Unfortunately, this once again collapses all of the way back down into a single weight matrix multiplied by each of the three inputs. It is the same linear model! We could continue this process ad infinitum and it would be the same result, albeit a lot more costly computationally for training or prediction from a much, much more complicated architecture than needed.

So how can you escape having a linear model? By adding non linearity of course :)

Adding a Non-Linearity



The solution is adding a nonlinear transformation layer which is facilitated by a nonlinear activation function such as Sigmoid, Tan-h, or ReLU. In thinking of terms of the graph that is created by TensorFlow, you can imagine each neuron actually having two nodes.

The first node being the result of the weighted sum $W * x + b$ and the second node being the result of that being passed through the activation function. In other words, there are the inputs to the activation function followed by the outputs of the activation function so the activation function acts as the transition point between layers.

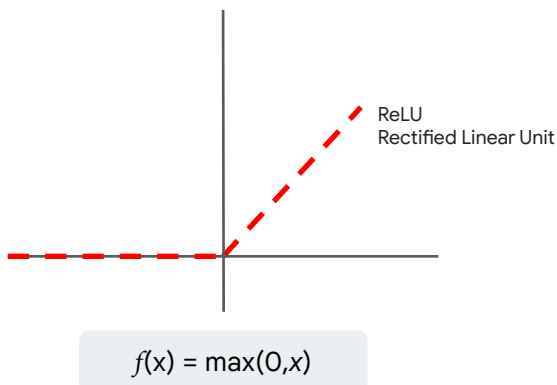
Adding in this nonlinear transformation is the only way to stop the neural network from condensing back into a shallow network.

Even if you have a layer with nonlinear activation functions in your network, if elsewhere in the network you have 2 or more layers with linear activation functions, those can still be collapsed into just one network.

Usually, neural networks have all layers nonlinear for the first $n - 1$ layers and then have the final layer transformation be linear (for regression) or sigmoid or softmax (for classification).

It all depends on what you want the output to be.

Our favorite non-linearity is the Rectified Linear Unit



There are many nonlinear activation functions with sigmoid, and the scaled and shifted sigmoid, hyperbolic tangent being some of the earliest. However, as I mentioned, these can have saturation which leads to the vanishing gradient problem where, with zero gradients, the model's weights don't update and training halts.

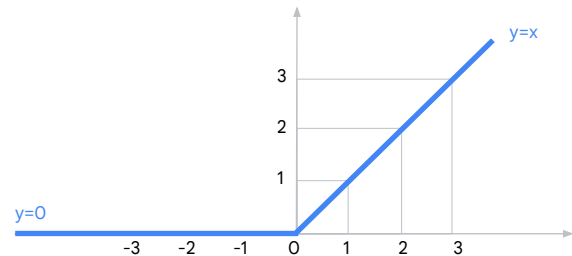
The Rectified Linear Unit, or ReLU for short, is one of our favorites because it's simple and works well. In the positive domain it is linear - so we don't have saturation - whereas in the negative domain the function is 0.

Networks with ReLU hidden activations often have 10 times the speed of training than networks with sigmoid hidden activations. However, due to the negative domain's function always being zero, we can end up with ReLU layers dying. What I mean by this is that when you start getting inputs in the negative domain then the output of the activation will be zero which doesn't help in the next layer in getting the inputs into the positive domain. This compounds and creates a lot of zero activations.

During backpropagation when updating the weights, since we have to multiply our error's derivative by the activation, we end up with a gradient of zero, thus a weight update of 0, and thus the weights don't change and training fails for that layer.

There are many
different **ReLU variants**

Normal ReLU activation function

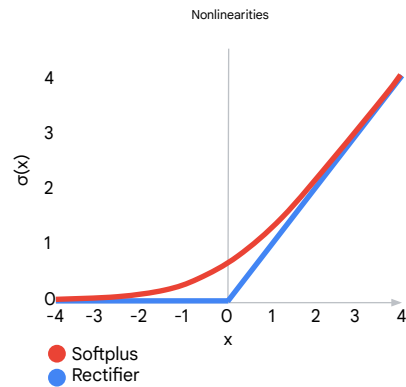


Fortunately, a lot of clever methods have been developed to slightly modify the ReLU and avoid the "dying ReLU" effect to ensure training doesn't stall, but still with much of the benefits of the vanilla ReLU.

Here again is the vanilla ReLU. The maximum operator can also be represented by the piecewise linear equation where less than 0 the function is 0 and greater than or equal to 0 the function is x.

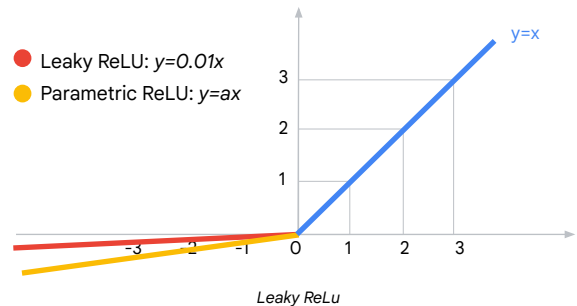
Some extensions to ReLU meant to relax the non-linear output of the function and allow small negative values are:

There are many
different **ReLU variants**



Softplus or SmoothReLU function. This function has as its derivative the logistic function. The logistic sigmoid function is a smooth approximation of the derivative of the rectifier.

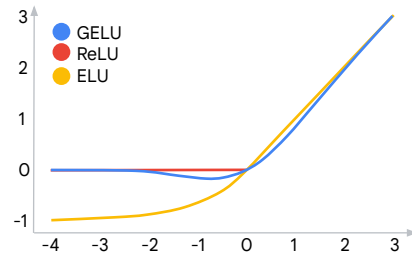
There are many
different **ReLU variants**



The Leaky ReLU function is modified to allow small negative values when the input is less than zero. Its rectifier allows for a small, non-zero gradient when the unit is saturated and not active.

The Parametric ReLU (PReLU) learns parameters that control the leakyness and shape of the function. It adaptively learns the parameters of the rectifiers.

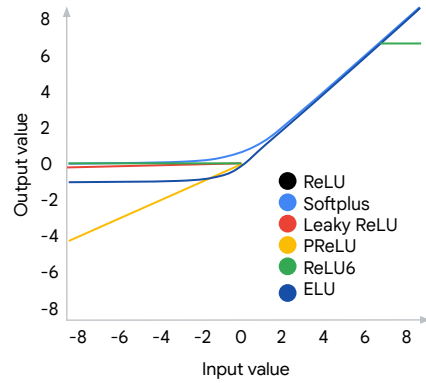
There are many
different **ReLU variants**



The Exponential Linear Unit, or ELU, is a generalization of the ReLU that uses a parameterized exponential function to transition from the positive to small negative values. Its negative values push the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient.

The Gaussian Error Linear Unit (GELU), is a high-performing neural network activation function. It's nonlinearity results in the expected transformation of a stochastic regularizer which randomly applies the identity or zero map to a neuron's input.

There are many
different **ReLU variants**



Here's a quick visualization overlay showing the most popular ReLU variants.

Keras is built-in to TF 2.x



- User-friendly
- Modular and composable
- Easy to extend

Tf.keras, again, that's TensorFlow's high-level API for building and training your deep learning models. It's also really useful for fast prototyping, state of the art research, and production lies in these models, and it has a couple of key advantages that you should be familiar with:

- It's user-friendly. Keras has a simple consistent interface optimized for your common ML use cases. It provides clear and actionable feedback for user errors, which makes it fun to write ML with.
- It's modular and composable Keras models are made together by connecting configurable building blocks together with just a few restrictions.
- Also, it's really easy to extend and write your own cost in building blocks to express new ideas on the leading edge of machine learning research. You can create new layers, create new metrics, loss-function, and develop your whole new state of the art machine learning model, should you wish.

Stacking layers with 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.

Here's an example. A sequential model, like you see here in code, is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. Sequential models are not really advisable if the model that you're building has multiple inputs or multiple outputs. Any of the layers in the model have multiple inputs and multiple outputs. The model needs to do layer sharing or the model has a non-linear topology, such as a residual connection or if it is multi-branches.

Let's look at some more code.


```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

A linear model (multiclass
logistic regression)

In this example, you'll see that there's one single dense layer being defined. That layer has 10 nodes or neurons and the activation is a softmax. The activation being softmax tells us we're probably doing classification.

```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```


A linear model (multiclass
logistic regression)

A linear model (a single Dense
layer) aka multiclass logistic
regression

With a single layer, the model is linear. This example is able to perform logistic regression and classify examples across 10 classes.

```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```




A neural network with one hidden layer
A neural network with one hidden layer

With the addition of another dense layer, the model now becomes a neural network with one hidden layer. But it's possible to map nonlinearities through that ReLU activation, we talked about before.

```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



A neural network with multiple hidden layers (a deep neural network)

Once more, you're going to add one layer to the network. Now it's becoming a deeper neural network. Each additional layer makes it deeper and deeper and deeper. Now, let's try that again.

```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



A deeper neural network

Here's another deeper neural network architecture. Needless to say, the deeper a neural net gets, generally the more powerful it becomes in learning patterns from your data. But one thing you really have to watch out for, is this can cause the model to over fit as it may almost learn all the patterns in the data by memorizing it and not generalize to unseen data.

Now there are mechanisms to avoid that like regularization, and we'll talk about those later.

Compiling a Keras model

```
def rmse(y_true, y_pred):  
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))  
  
model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```

Custom Metric

Loss function

Once we define the model object, we compile it. During model compilation, a set of additional parameters are passed to the method. These parameters will determine the optimizer that should be used, the loss function, and the evaluation metrics. Other parameter options could be the loss weight, the sample weight mode, and the weighted metrics if you get really advanced into this.

What is a loss function? Well, that's your guide to the terrain telling the optimizer when it's moving in the right or wrong direction for reducing the loss.

Compiling a Keras model

```
def rmse(y_true, y_pred):  
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))  
  
model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```

Optimizer

Optimizers tied together that loss function and the model parameters by actually doing the updating of the model in response to the output of the loss function. In plain terms, optimizers shape and mold your model into its most accurate possible form by playing around with those weights.

An optimizer that is generally used in machine learning is SGD or stochastic gradient descent. SGD is an algorithm that descends the slope, hence the name, to reach the lowest point on that loss surface. A useful way to think of this, is think of that surface is a graphical representation of the data in the lowest point in that graph, as where that error is at a minimum.

Optimizers aim to take the model there through successive training runs.

Compiling a Keras model

```
def rmse(y_true, y_pred):  
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))  
  
model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])
```

In this example, the optimizer that we're using is called Adam. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on the training data.

The algorithm is straightforward to implement. Besides being computationally efficient and having little memory requirements, another advantage of Adam, is its invariability due to the diagonal re-scaling of the gradients. Adam is well-suited for models that have a large and large and large datasets, or if you have a lot of parameters that you're adjusting.

The method is also very appropriate for problems with very noisy or sparse gradients and nonstationary objectives.

In case you're wondering, besides Adam, some additional optimizers are Momentum, which reduces learning rate when the gradient values are small. Adagrad, which gives frequently occurring features low learning rates. Adadelat improves Adagrad by avoiding and reducing LR to 0, and the last one, which is a pretty cool name, Ftrl, or follow the regularized leader. I love that name. It works well on wide models. At this time, Adam and Ftrl make really good defaults for deep neural nodes as well as linear models that you're building.

Training a Keras model

```
from tensorflow.keras.callbacks import TensorBoard
```

```
steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)
```

```
history = model.fit(  
    x=trainds,  
    steps_per_epoch=steps_per_epoch,  
    epochs=NUM_EVALS,  
    validation_data=evalds,  
    callbacks=[TensorBoard(LOGDIR)]  
)
```

This is a trick so that we have control on the total number of examples the model trains on (NUM_TRAIN_EXAMPLES) and the total number of evaluation we want to have during training (NUM_EVALS).

Now is the moment that we've all been waiting for, it's time to train the model that we just defined. We'll train models in Keras by calling the fit method.

You can pass parameters to fit that define the number of epochs. Again, an epoch is a complete pass on the entire training dataset. Steps for epoch, which is the number of batch iterations before a training epoch is considered finished. Validation data, validation steps, batch size, which determines the number of samples in each mini-batch. It's maximum is the number of all samples and others such as callbacks.

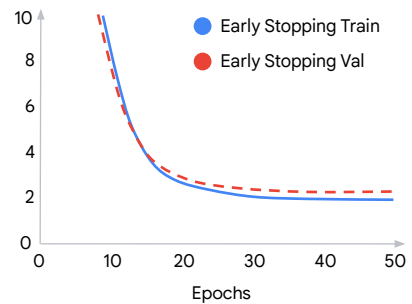
Callbacks are utilities called at certain points during model training for activities such as logging and visualization using tools such as TensorBoard.

Training a Keras model

```
from tensorflow.keras.callbacks import TensorBoard

steps_per_epoch = NUM_TRAIN_EXAMPLES //
(TRAIN_BATCH_SIZE * NUM_EVALS)

history = model.fit(
    x=trainds,
    steps_per_epoch=steps_per_epoch,
    epochs=NUM_EVALS,
    validation_data=evalds,
    callbacks=[TensorBoard(LOGDIR)]
)
```



Saving the training iterations to a variable allows for plotting of all your chosen evaluation metrics like mean absolute error, root mean squared error, accuracy, etc, versus the epochs for example, like you see here.

```
%tensorflow_version 2.x
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.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, 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)
```



A deeper neural network

Here's a code snippet with all of the steps put together. The model definition, compilation, fitting, and evaluation.

Once trained, the model can be used for prediction

```
predictions = model.predict(input_samples, steps=1)
```

returns a Numpy array of predictions

With the predict() method you can pass

- a Dataset instance
- Numpy array
- a TensorFlow tensor, or list of tensors
- a generator of input samples

steps determines the total number of steps before declaring the prediction round finished. Here, since we have just one example, steps=1.

Once trained, the model can now be used for predictions or inferences. You'll need an input function that provides data for the prediction. So back to our example of the housing price model. We could predict the house prices with examples of a 1500 square foot house and an 1800 square foot apartment, for example. The predict function in a tf.keras API returns a Numpy array or arrays of the predictions.

The steps parameter determines the total number of steps before declaring the prediction round finished. Here since we have just one example, we set steps equal to 1. Setting steps equal to none would also work here. Note however, that if the input samples and the tf.data.dataset or a dataset iterator and steps is set to none, predictable run into the input dataset is exhausted.

**To serve our model for others to use, we
export the model file and deploy
the [model as a service](#).**

Once you have selected the features, transformed them as needed, chosen your model architecture, applied any regularization that is necessary to ensure good performance, trained your model, and iterated through this process a couple of times, it is time to serve the model for prediction.

Of course, making individual predictions is not realistic, because we can't expect client code to have a model object in memory. For others to use our trained model, we'll have to save (or export) our model to a file, and expect client code to instantiate the model from that exported file.

We'll export the model to a TensorFlow SavedModel format. Once we have a model in this format, we have lots of ways to "serve" the model, from a web application, from JavaScript, from mobile applications, etc.

SavedModel is the universal serialization format for TensorFlow models

```
OUTPUT_DIR = "./export/savedmodel"
shutil.rmtree(OUTPUT_DIR, ignore_errors=True)

EXPORT_PATH = os.path.join(OUTPUT_DIR,
datetime.datetime.now().strftime("%Y%m%d%H%M%S"))

tf.saved_model.save(model, EXPORT_PATH)
```

exports a model object to a
SavedModel format

a trackable object such as a
trained keras model

the directory in which to
write the SavedModel

SavedModel is the universal serialization format for TensorFlow models.

SavedModel provides a language-neutral format to save machine-learned models that is recoverable and hermetic. It enables higher-level systems and tools to produce, consume and transform TensorFlow models.

The resulting SavedModel is then servable.

Models saved in this format can be restored using `tf.keras.models.load_model` and are compatible with TensorFlow Serving.

Create a model object in Vertex AI

```
MODEL_NAME=propertyprice
VERSION_NAME=dnn

if [[ $(gcloud ai-platform models list --format='value(name)' | grep $MODEL_NAME) ]];
then
    echo "$MODEL_NAME already exists"
else
    echo "Creating $MODEL_NAME"
    gcloud ai-platform models create --regions=$REGION $MODEL_NAME
fi
...
```

create the model in AI Platform

One of the ways we can serve the model is to utilize the AI Platform managed service. The AI Platform service also performs scaled training, but for now we are focusing on serving a trained model.

You start by creating a model object in AI Platform.

Will call our model 'propertyprice'.

Create a version of the model in Vertex AI

```
MODEL_NAME=propertyprice
VERSION_NAME=dnn

...

if [[ $(gcloud ai-platform versions list --model $MODEL_NAME --format='value(name)' |
grep $VERSION_NAME) ]]; then
    echo "Deleting already existing $MODEL_NAME:$VERSION_NAME ..."
    echo yes | gcloud ai-platform versions delete --model=$MODEL_NAME $VERSION_NAME
    echo "Please run this cell again if you don't see a Creating message ..."
    sleep 2
fi
```

create the model version in AI Platform

Next, we need to create a version for our model

Will call our version 'dnn'. You can also utilize timestamp or another differentiator for multiple versions of the same model type.

Deploy SavedModel using gcloud ai-platform

```
gcloud ai-platform versions create \  
  --model=$MODEL_NAME $VERSION_NAME \  
  --framework=tensorflow \  
  --python-version=3.5 \  
  --runtime-version=2.1 \  
  --origin=$EXPORT_PATH \  
  --staging-bucket=gs://$BUCKET
```

specify the model name
and version

EXPORT_PATH denotes
location of SavedModel
directory

Once model and version are created, you can run this command to push the model to the cloud.

Remember to point to the output directory in which the SavedModel was saved to.

The command to push the model also takes other flags such as python and tensorflow runtime versions, the framework (in case you are using Scikit learn or xgboost - this flag defaults to TensorFlow), and a bucket in which to stage training archives.

A staging bucket is required only if a file upload is necessary (that is, other flags include local paths).

Make predictions using gcloud ai-platform

```
input.json = {"sq_footage": 3140,  
              "type": 'house'}
```

```
gcloud ai-platform predict \  
  --model propertyprice \  
  --version dnn \  
  --json-instances input.json
```

specify the name and version
of the deployed model

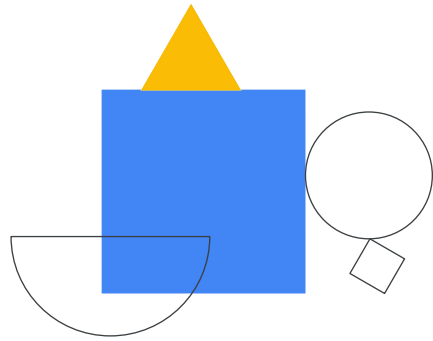
json file for prediction

Once the model is created and pushed to AI Platform, use the `gcloud ai-platform predict` command to perform predictions.

Make sure that the flags include the model name, its version, and the path to a file containing the examples you want to get predictions on.

Lab intro

Introducing the Keras Sequential API



The Keras sequential API allows you to create Tensorflow models layer-by-layer. This is useful for building most kinds of machine learning models but it does not allow you to create models that share layers, re-use layers or have multiple inputs or outputs.

Lab objectives

1

Build a DNN model using the Keras Sequential API

2

Use feature columns in a Keras model

3

Train a model with Keras

4

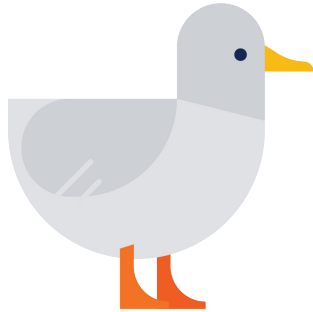
Save, load, and deploy a Keras model on Google Cloud

5

Deploy and make predictions with a Keras model

In this lab, we'll see how to build a simple deep neural network model using the keras sequential api and feature columns.

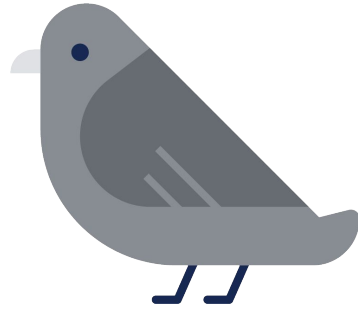
Once we have trained our model, we will deploy it using Vertex AI and see how to call our model for online prediction.



Seagulls can fly

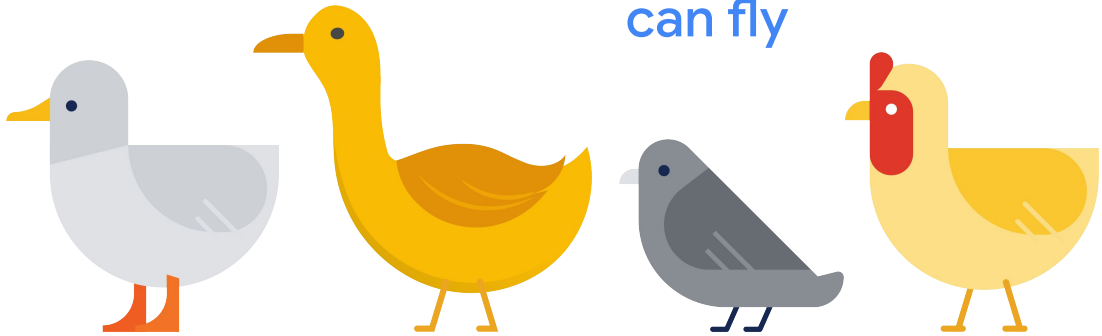
No, this is not a section about ornithology. But we all know that seagulls can fly, right?

Pigeons can fly

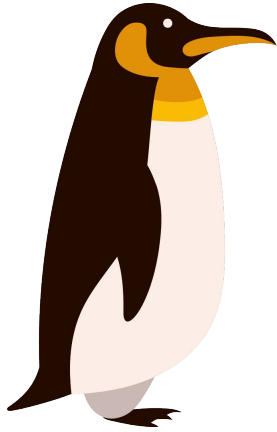


We also know that pigeons can fly as well.

Animals with wings
can fly



It is intuitive that animals with wings can fly, so making that generalization feels kinda natural.

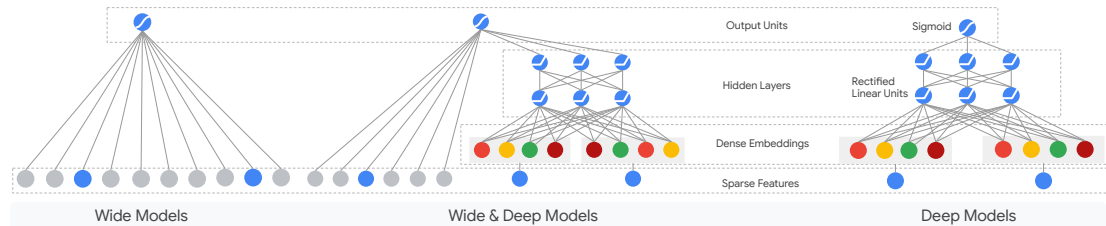


Penguins...

But what about penguins?

Using Wide and Deep learning

“ Combine the power of memorization and generalization on one unified machine learning model. ”

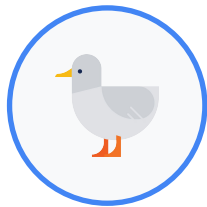


It's not an easy question to answer, but by jointly training a wide linear model (for **memorization**) alongside a deep neural network (for **generalization**), one can combine the strengths of both to bring us one step closer.

At Google, we call it Wide & Deep Learning. It's useful for generic large-scale regression and classification problems with sparse inputs (categorical features with a large number of possible feature values), such as recommender systems, search, and ranking problems.

Memorization + Generalization

- Memorization: “Seagulls can fly.” “Pigeons can fly.”
- Generalization: “Animals with wings can fly.”
- Generalization + memorizing exceptions: “Animals with wings can fly, but penguins cannot fly.”



The human brain is a sophisticated learning machine, forming rules by memorizing everyday events (“seagulls can fly” and “pigeons can fly”) and generalizing those learnings to apply to things we haven’t seen before (“animals with wings can fly”).

Perhaps more powerfully, memorization also allows us to further refine our generalized rules with exceptions (“penguins can’t fly”).

As we were exploring how to advance machine intelligence, we asked ourselves the question—can we teach computers to learn like humans do, by combining the power of memorization and generalization?

Linear models are
good for sparse,
independent features

```
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]  
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]  
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]  
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]  
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
```

This is what a sparse matrix looks like -- very, very wide, with lots and lots of features.

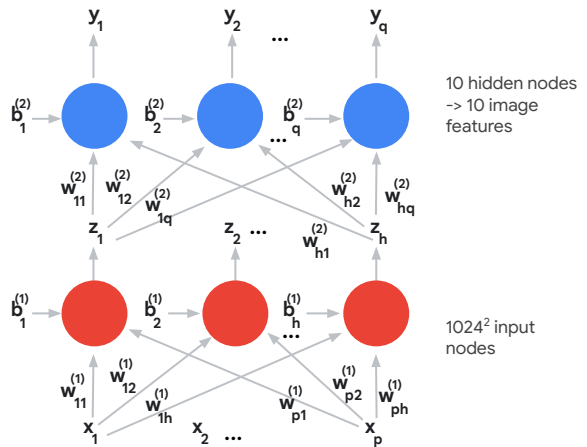
You want to use linear models to minimize the number of free parameters. And if the columns are independent, linear models may suffice.

DNNs are good for dense,
highly correlated
features

pixel_values (

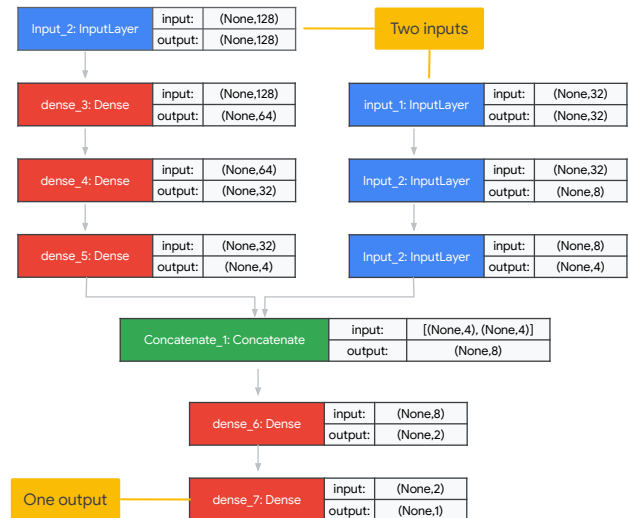


)



Nearby pixels, however, tend to be highly correlated, so putting them through a NN, we have the possibility that the inputs get decorrelated and mapped to a lower dimension (intuitively, this is what happens when your input layer takes each pixel value, and the number of hidden nodes is much less than the number of input nodes).

Wide and deep networks using Keras Functional API

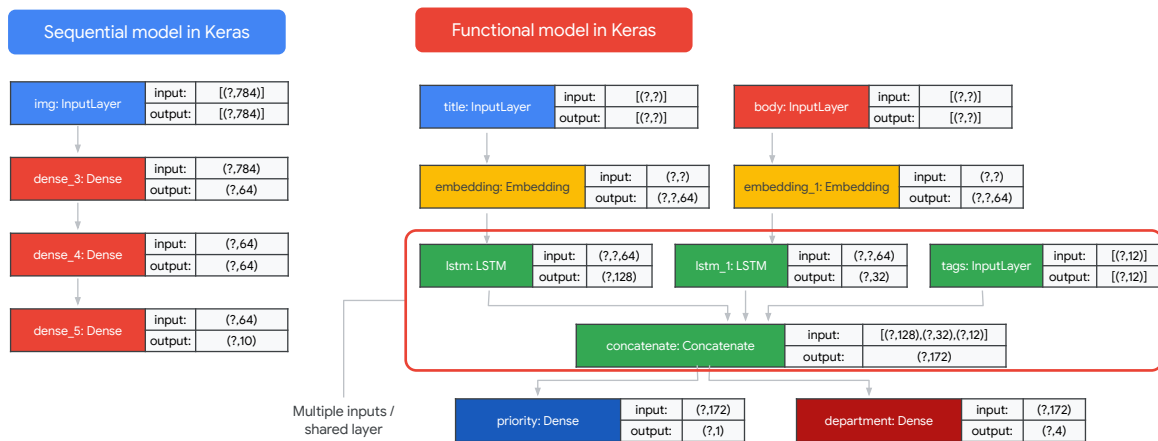


A Wide and Deep model architecture is an example of a complex model that can be built using the Keras Functional API.

The Functional API gives the model the ability to have multiple inputs and outputs. It also allows for models to share layers. Actually, more than that, it allows you to define ad hoc acyclic network graphs.

With that functional API, models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model.

Functional API is more flexible than Sequential API



The Functional API is a way to create models that is more flexible than Sequential: it can handle models with non-linear topology, models with shared layers, and models with multiple inputs or outputs.

The functional API makes it easy to manipulate multiple inputs and outputs. This cannot be handled with the Sequential API.

Here's a simple example. Let's say you're building a system for ranking custom issue tickets by priority and routing them to the right department.

Your model will have **four** inputs:

- Title of the ticket (text input)
- Text body of the ticket (text input)
- Any tags added by the user (categorical input)
- An image representing different logos that can appear on the ticket

It will have **two** outputs:

- The department that should handle the ticket (softmax output over the set of departments)
- A text sequence with a summary of the text body

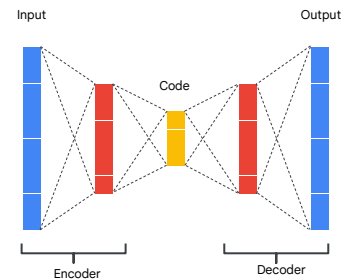
Models are created by specifying their inputs and outputs in a graph of layers

```
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Dense(16, activation='relu')(encoder_input)
x = layers.Dense(10, activation='relu')(x)
x = layers.Dense(5, activation='relu')(x)
encoder_output = layers.Dense(3, activation='relu')(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')

x = layers.Dense(5, activation='relu')(encoder_output)
x = layers.Dense(10, activation='relu')(x)
x = layers.Dense(16, activation='relu')(x)
decoder_output = layers.Dense(28, activation='linear')(x)

autoencoder = keras.Model(encoder_input, decoder_output, name='autoencoder')
```



In the functional API, models are created by specifying their inputs and outputs in a graph of layers. That means that a single graph of layers can be used to generate multiple models. You can treat any model as if it were a layer, by calling it on an input or on the output of another layer. Note that by calling a model you aren't just reusing the architecture of the model, you're also reusing its weights.

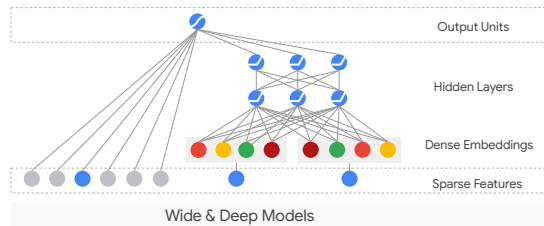
This is an example of what code for an autoencoder might look like. Notice how the operations are treated like functions with the outputs serving as inputs for subsequent layers.

Another good use for the functional API are models that use shared layers. Shared layers are layer instances that get reused multiple times in a same model: they learn features that correspond to multiple paths in the graph-of-layers. Shared layers are often used to encode inputs that come from similar spaces (say, two different pieces of text that feature similar vocabulary), since they enable sharing of information across these different inputs, and they make it possible to train such a model on less data. If a given word is seen in one of the inputs, that will benefit the processing of all inputs that go through the shared layer.

To share a layer in the Functional API, just call the same layer instance multiple times.

Creating a Wide and Deep model in Keras

```
INPUT_COLS = [  
    'pickup_longitude',  
    'pickup_latitude',  
    'dropoff_longitude',  
    'dropoff_latitude',  
    'passenger_count'  
]  
  
# Prepare input feature columns  
inputs = {colname : layers.Input(name=colname,  
    shape=(), dtype='float32')  
    for colname in INPUT_COLS  
}  
...
```



To create a wide and deep model in Keras, start by setting up the input layers for the model, using the features of the model data.

For this example, we are using pickup and dropoff latitude and longitude, as well as the number of passengers, to try and predict the taxi fare for a ride.

These inputs will be fed to the wide and deep portions of the model.

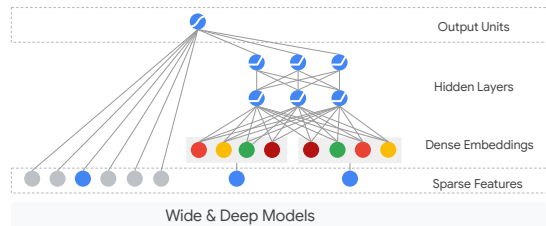
Creating a Wide and Deep model in Keras

```
# Create deep columns
deep_columns = [
    # Embedding_column to "group" together ...
    fc.embedding_column(fc_crossed_pd_pair, 10),

    # Numeric columns
    fc.numeric_column("pickup_latitude"),
    fc.numeric_column("pickup_longitude"),
    fc.numeric_column("dropoff_longitude"),
    fc.numeric_column("dropoff_latitude")]

# Create the deep part of model
deep_inputs = layers.DenseFeatures
    (deep_columns, name='deep_inputs')(inputs)
x = layers.Dense(30, activation='relu')(deep_inputs)
x = layers.Dense(20, activation='relu')(x)

deep_output = layers.Dense(10, activation='relu')(x)
```



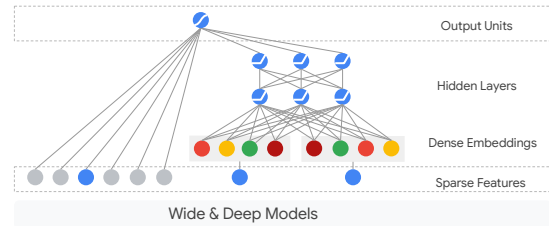
Using the inputs above, we can then create the deep portion of the model. `layers.Dense` is a densely-connected NN layer. By stacking multiple layers, we make it deep.

Creating a Wide and Deep model in Keras

```
# Create wide columns
wide_columns = [
    # One-hot encoded feature crosses
    fc.indicator_column(fc_crossed_dloc),
    fc.indicator_column(fc_crossed_ploc),
    fc.indicator_column(fc_crossed_pd_pair)
]
```

```
# Create the wide part of model
wide = layers.DenseFeatures(wide_columns,
                             name='wide_inputs')(inputs)
```

created in the previous slide



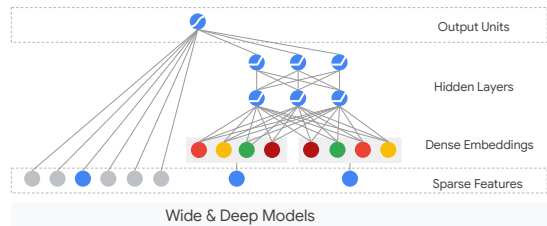
We can also create the wide portion of the model using, for example, `DenseFeatures`, which produces a dense Tensor based on given `feature_columns`.

Creating a Wide and Deep model in Keras

```
# Combine outputs
combined = concatenate(inputs=[deep, wide],
                       name='combined')
output = layers.Dense(1,
                      activation=None,
                      name='prediction')(combined)

# Finalize model
model = keras.Model(inputs=list(inputs.values()),
                    outputs=output,
                    name='wide_and_deep')
model.compile(optimizer="adam",
              loss="mse",
              metrics=[rmse, "mse"])
```

To finalize the model, specify
the inputs and outputs



Lastly we combine the wide and deep portions and compile the model.

Training, evaluation, and inference work exactly in the same way for models built using the Functional API as for Sequential models.

Strengths and weaknesses of the Functional API

Strengths

- Less verbose than using keras.Model subclasses
- Validates your model while you're defining it
- Your model is plottable and inspectable
- Your model can be serialized or cloned

Weaknesses

- Doesn't support dynamic architectures
- Sometimes you have to write from scratch and you need to build subclasses, e.g. custom training or inference layers

Strengths

- It is less verbose than using keras.Model subclasses.
- It validates your model while you're defining it. In the Functional API, your input specification (shape and dtype) is created in advance (via Input), and every time you call a layer, the layer checks that the specification passed to it matches its assumptions, and it will raise a helpful error message if not. This guarantees that any model you can build with the Functional API will run. All debugging (other than convergence-related debugging) will happen statically during the model construction, and not at execution time. This is similar to typechecking in a compiler.
- Your Functional model is plottable and inspectable.
- You can plot the model as a graph, and you can easily access intermediate nodes in this graph -- for instance, to extract and reuse the activations of intermediate layers. Your Functional model can be serialized or cloned. Because a Functional model is a data structure rather than a piece of code, it is safely serializable and can be saved as a single file that allows you to recreate the exact same model without having access to any of the original code. See our saving and serialization guide for more details.

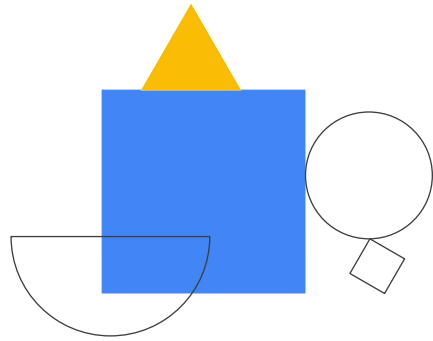
Weaknesses

- It does not support dynamic architectures. The Functional API treats models as DAGs of layers. This is true for most deep learning architectures, but not all: for instance, recursive networks or Tree RNNs do not follow this assumption and cannot be implemented in the Functional API.

- Sometimes, you just need to write everything from scratch. When writing advanced architectures, you may want to do things that are outside the scope of "defining a DAG of layers": for instance, you may want to expose multiple custom training and inference methods on your model instance. This requires subclassing.

Lab intro

Build a DNN using the Keras Functional API



In this lab, you will export BigQuery data to GCS to train a Keras model.

Lab objectives

- 1 Review how to read in CSV file data using `tf.data`
- 2 Specify input, hidden, and output layers in the DNN architecture
- 3 Train the model locally and visualize the loss curves
- 4 Deploy and predict with the model using Cloud AI Platform

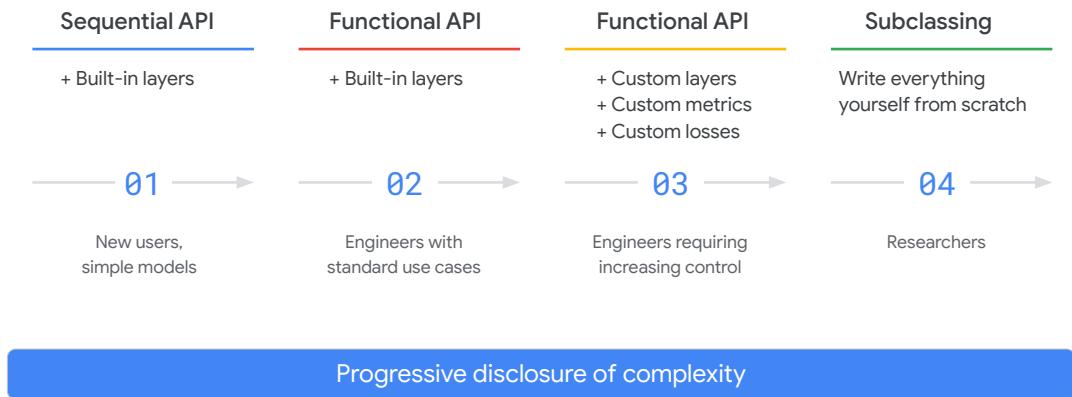
First, you'll review how to read in CSV file data using `tf.data`.

Then specify input, hidden, and output layers in the DNN architecture.

After that train the model locally and visualize the loss curves.

Finally, deploy and predict with the model using Vertex AI.

Model building: From simple to arbitrarily flexible



Let's start with where model subclassing fits in the spectrum of module building. We've already looked at two Keras models.

- The **Sequential model** is very straightforward and consists of a simple list of layers. But it is limited to single-input, single-output stacks of layers, as the name suggests.
- The **Functional API** is an easy-to-use, fully featured API that supports arbitrary model architectures. Most people, in most use cases, should use the Functional API, which is the Keras "industry strength" model.
- The third way to create a Keras model is **model subclassing**, where you implement everything from scratch on your own. You should use this if you have complex, out-of-the-box research use cases.

Choosing a model depends on the amount of customization you need. The Sequential model does not allow you much flexibility in creating your models. Although more flexible than the sequential model, the Functional API also has limits on customization. With model subclassing, you can create your own fully customizable models in Keras. This is done by subclassing the Model class and implementing a call method. Let's consider some examples.

Define layers

```
import tensorflow as tf

class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

    def call(self, inputs):
        x = self.dense1(inputs)
        return self.dense2(x)

model = MyModel()
```

By subclassing the Model class, you define your layers in `__init__` (or the “initializer”) and you implement the model's forward pass

Implement the forward pass

```
import tensorflow as tf

class MyModel(tf.keras.Model):

    def _init_(self):
        super(MyModel, self)._init_()
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)

    def call(self, inputs):
        x = self.dense1(inputs)
        return self.dense2(x)

model = MyModel()
```

in the call method.

Constructor method

```
import tensorflow as tf

class MyModel(tf.keras.Model):

    def _init_(self):
        super(MyModel, self)._init_()
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)
```

Constructor

So, when you use model subclassing, you have a constructor method, which defines the layers to use; in this example, a pair of dense layers.

Call method

```
def call(self, inputs):  
    x = self.dense1(inputs)  
    return self.dense2(x)
```

```
model = MyModel()
```

Call method

You also have the call method, which orders the layers. In our example, the call method feeds the inputs through the dense layer and then feeds that result through the second dense layer.

The “constructor” and “call method”

```
import tensorflow as tf
```

```
class MyModel(tf.keras.Model):
```

```
    def __init__(self):
```

```
        super(MyModel, self).__init__()
```

```
        self.dense1 = tf.keras.layers.Dense(4, activation=tf.nn.relu)
```

```
        self.dense2 = tf.keras.layers.Dense(5, activation=tf.nn.softmax)
```

```
    def call(self, inputs):
```

```
        x = self.dense1(inputs)
```

```
        return self.dense2(x)
```

```
model = MyModel()
```

Subclass the model
directly

Constructor

Call method

The model is subclassed directly where MyModel inherits from tf.keras.Model.

The “constructor” and “call method”

```
class MyModel(tf.keras.Model):  
    def __init__(self, num_classes=10):  
        super(MyModel, self).__init__(name='my_model')  
        # Define your layers here.  
        self.dense_1 = layers.Dense(32, activation='relu')  
        self.dense_2 = layers.Dense(num_classes, activation='softmax')  
  
    def call(self, inputs):  
        # Define your forward pass here  
        # using layers you previously defined in '__init__'  
        x = self.dense_1(inputs)  
        return self.dense_2(x)
```

Subclass the model
directly

Constructor

Call method

Indeed, model subclassing is fully customizable. Let's consider another way you can use model subclassing to further customize this example.

One way is to define the number of classes—which is an extra argument in the constructor—so the user can set the number of classes the model is predicting.

This argument is used in the second dense layer to determine the number of neurons in this layer.

For complete flexibility, consider a customer training loop.

Custom training loop

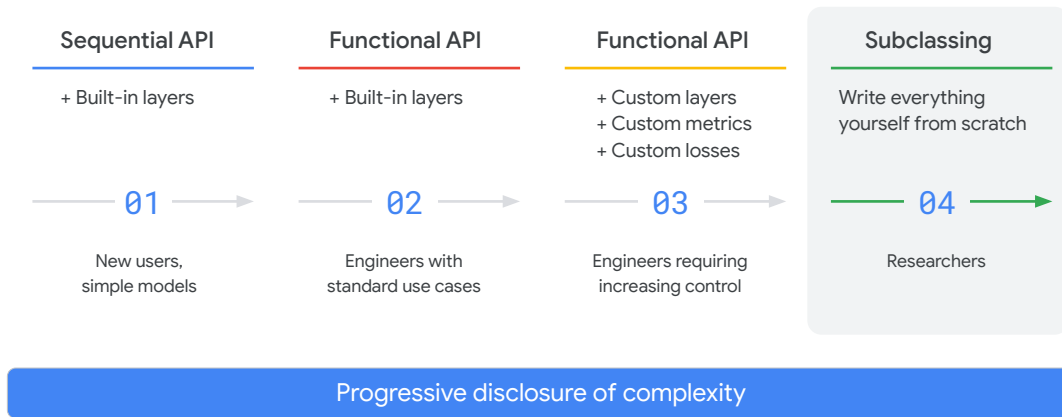
```
model = MyModel()
with tf.GradientTape() as tape:
    logits = model(images, training=True)
    loss_value = loss(logits, labels)
grads = tape.gradient(loss_value, model.variables)
optimizer.apply_gradients(zip(grads, model.variables))
```

The Sequential, Functional, and Model Subclass model types can be compiled and trained using the simple compile and fit commands, or you can write your own custom training loop for complete control.

For example, you can use the training keyword argument to determine the behavior of the model at training time and at test time. This keyword argument should be a boolean (either true or false). A really common use of this keyword argument is in batch normalization and dropout layers, because some neural network layers behave differently during training and inference.

For example, during training, dropout will randomly drop out units and correspondingly scale up activations of the remaining units. During inference, it does nothing (since you usually don't want the randomness of dropping out units here).

Model building: From simple to arbitrarily flexible

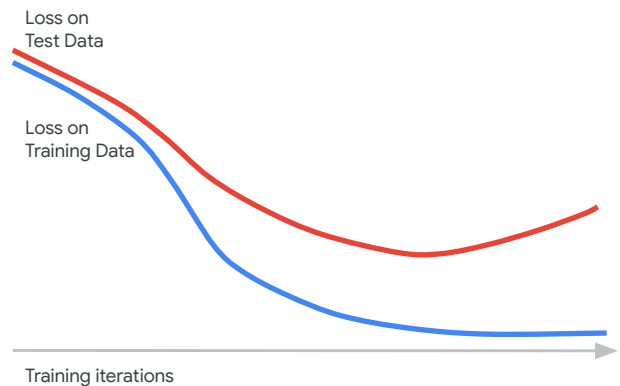


Model subclassing is at the end of the spectrum of model building in Keras.

Although you can use the Sequential API for ease of use or the Functional API for more customization, use Model Subclassing for complete flexibility and control.

What is
happening here?

How can we
address this?



Remember our goal while training a model is to minimize the loss value. If you graphed the loss curve both on training and test data, it may look something like this. The graph shows Loss on the y axis vs. Time on the x axis.

Notice anything wrong here?

Yeah, the loss value is nicely trending down on the training data but shoots upwards at some point on the test data. That cannot be good!

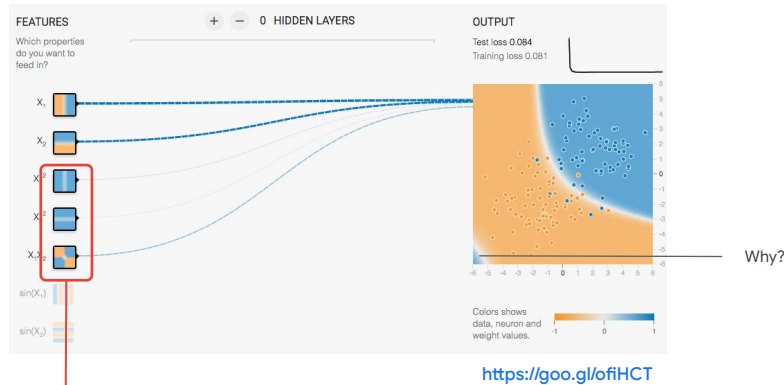
Clearly, some amount of overfitting is going on here. Seems to be correlated with the number of training iterations. How could we address this?

We could reduce number of training iterations and stop earlier. Early stopping is definitely an option, but there must be better ones...

Here is where Regularization comes into the picture!

Remember the splotch of blue?

Why does it happen?



Is the model behavior surprising? What's the issue?
Try removing cross-product features. Does performance improve?

Let's tickle our intuition using TensorFlow playground.

You must have seen and used this playground in previous courses, but to quickly remind you: TensorFlow Playground is a handy little tool for visualizing how neural networks learn. We extensively use it throughout this specialization to intuitively grasp the concepts.

Let me draw your attention to the screen. There is something odd going on here. Notice this region in the bottom left that's hinting towards blue? There is nothing in the data suggesting blue.

The model's decision boundary is kind of crazy! Why do you think that is? Notice the relative thickness of the five lines running from INPUT to OUTPUT.

These lines show the relative weights of the five features. The lines emanating from X_1 and X_2 are much thicker than those coming from the feature crosses. So, the feature crosses are contributing far less to the model than the normal (uncrossed) features. Removing all the feature crosses gives a saner model.

You should try this for yourself and see how curved boundary suggestive of overfitting disappears and test loss converges.

After 1,000 iterations, test loss should be a slightly lower value than when the feature crosses were in play. Although your results may vary a bit, depending on the data set.

The data in this exercise is basically linear data plus noise. If we use a model that is too complicated, such as one with too many crosses, we give it the opportunity to fit to the noise in the training data, often at the cost of making the model perform badly on test data.

Clearly, early stopping cannot help us here. It's the model complexity that we need to bring under control. But, how could we measure model complexity and avoid it?

Occam's razor

When presented with competing hypothetical answers to a problem, one should select the one that makes the fewest assumptions.

The idea is attributed to William of Ockham (c. 1287–1347).



There is a whole field around this called Generalization Theory or G Theory that goes about defining the statistical framework.

The easiest way to think about it, though, is by intuition, based on 14th century principle laid out by William Ockham.

While training a model we will apply Okham's razor principle as our heuristic guide in favoring simpler models with less assumptions about the training data.

Let's look into some of the most common Regularization techniques that can help us apply this principle in practice.

Factor in model complexity when calculating error

Minimize: $\text{loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model})$

aim for low
training error

...but balance against
complexity

Optimal model complexity is data-dependent, so requires hyperparameter tuning.

The idea is to penalize model complexity.

So far in our training process, we have been trying to minimize loss of the data, given the model; we need to balance that against complexity of the model.

Before we talk about how to measure model complexity, let's pause and understand why we said "balance" complexity against loss.

The truth is that oversimplified models are useless! If we take it to extreme, we will end up with a null model. We need to find the right balance between simplicity and accurate fitting of training data.

Later you will see that the complexity measure is multiplied by a lambda coefficient, which will allow us to control our emphasis on model simplicity. This makes up yet another hyperparameter that requires tuning.

Optimal lambda value for any given problem is data dependent, which means we almost always need to spend some time tuning this (either manually or via automated search). I told you ML needs some artful skills!

I hope by now it is clear why this approach is arguably more principled than early stopping.

Regularization is a major field of ML research

```
graph LR; A[Early Stopping] --- B[Parameter Norm Penalties]; B --- C[L1 regularization]; B --- D[L2 regularization]; B --- E[Max-norm regularization]; C --- F[Dataset Augmentation]; C --- G[Noise Robustness]; C --- H[Sparse Representations]; C --- I[...];
```

Early Stopping
Parameter Norm Penalties
 L1 regularization
 L2 regularization
 Max-norm regularization
Dataset Augmentation
Noise Robustness
Sparse Representations
...

We will look into these methods.

Regularization is one of the major fields of research within Machine Learning.

There are many published techniques and more to come:

- We already mentioned “Early Stopping”.
- We also started exploring the group of methods under the umbrella of “Parameter Norm Penalties”.
- There is also “Dataset Augmentation” methods, “Noise Robustness”, “Sparse Representations” and many more.

In this module, we will have a closer look at L1 and L2 regularization methods from “Parameter Norm Penalties” group of techniques.

But, before we do that, let’s quickly remind ourselves what problem “Regularization” is solving for us: Regularization refers to any technique that helps generalize a model. A generalized model performs well not just on training data, but also on never-seen test data.

How can we measure model complexity?

We know we are going to use regularization methods that penalize model complexity.

Now, the question is how to measure model complexity?

Both L1 and L2 regularization methods represent model complexity as the magnitude of the weight vector and try to keep that in check.

From Linear Algebra, you should remember that the magnitude of a vector is represented by the Norm function.

Let's quickly review L1 and L2 Norm functions.

L2 vs. L1 Norm

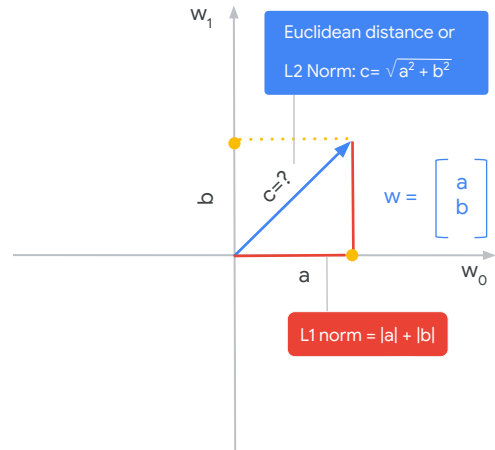
$$w = [w_0, w_1, \dots, w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + \dots + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + \dots + |w_n|)$$

L1 norm



- The weight vector can be of any number of dimensions, but it's easier to visualize in 2-dimensional space. So, a vector with $w_0=a$ and $w_1=b$ would look like this blue arrow. Now, what's the magnitude of this vector?
- You may instantly think c ? Because you are applying the most common way that we learnt in high school: the Euclidean distance from the origin. c would be the square root of sum of **a**-squared plus **b**-squared.
- In Linear Algebra, this is called the L2 norm: denoted by the double bars and the subscript of 2, or no subscript at all, because 2 is the known default. The L2 norm is calculated as the square root of sum of the squared values of all vector components. But that's not the only way the magnitude of a vector can be calculated.
- Another common method is L1 norm. L1 measures absolute value of **a** plus absolute value of **b**. Basically, the yellow path highlighted here. Now, remember we were looking for a way to define model complexity. We used L1 and L2 as regularization methods where model complexity is measured in the form of magnitude of the weight vector. In other words, if we keep the magnitude of our weight vector smaller than certain value, we've achieved our goal.

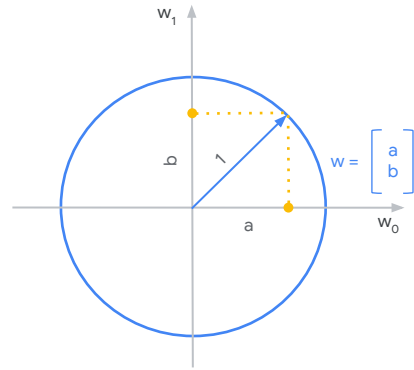
L2 vs. L1 Norm

$$w = [w_0, w_1, \dots, w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + \dots + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + \dots + |w_n|)$$



Now let's visualize what it means for the L2 norm of our weight vector to be under certain value, let's say 1.

Since L2 is the Euclidean distance from the origin, our desired vector would be bound within this circle with a radius of 1 centered on the origin.

L2 vs. L1 Norm

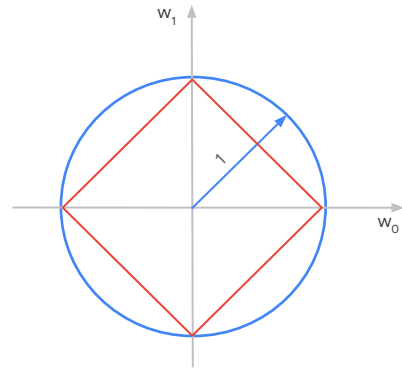
$$w = [w_0, w_1, \dots, w_n]^T$$

$$\|w\|_2 = [w_0^2 + w_1^2 + \dots + w_n^2]^{1/2}$$

L2 norm

$$\|w\|_1 = (|w_0| + |w_1| + \dots + |w_n|)$$

L1 norm



When trying to keep L1 norm under certain value, the area in which our weight vector can reside will take the shape of this yellow diamond.

The most important takeaway here is that when applying L1 regularization, the optimal value of certain weights can end up being zero.

And that is because of the extreme diamond shape of the optimal region that we are interested in.

That is as opposed to the smooth circular shape in L2 regularization.

In **L2 regularization**, complexity of model is defined by the **L2 norm** of the weight vector

Aim for low training error

...but balance against complexity

$L(w, D) + \lambda \|w\|_2$

Lambda controls how these are balanced

The diagram shows a blue rounded rectangle containing the equation $L(w, D) + \lambda \|w\|_2$. Three lines point to parts of the equation: one from 'Aim for low training error' to $L(w, D)$, one from '...but balance against complexity' to λ , and one from 'Lambda controls how these are balanced' to λ .

Let's go back to the problem at hand: how to regularize our model using vector norm.

This is how we apply L2 regularization, also known as weight decay. Remember we try to keep the weight values close to the origin. In 2D space the weight vector would be confined within a circle, you can easily expand the concept to 3D space, but beyond 3D is hard to visualize, don't try!

To be perfectly honest, in machine learning we cheat a little in the math department. We use the square of the L_2 norm to simplify calculation of derivatives.

Notice there's a new parameter here: lambda. This is a simple scalar value that allows us to control how much emphasis we want to put on model simplicity over minimizing training error.

It is another tuning parameter which must be explicitly set. Unfortunately, the best value for any given problem is data dependent. So, we'll need to do some tuning, either manually or automatically using a tool like hyperparameter tuning (which we will cover in the next module).

In L1 regularization,
complexity of model is
defined by the L1 norm
of the weight vector

$$L(w, D) + \lambda \|w\|_1$$

L1 regularization can be used as a
feature selection mechanism.

To apply L1 regularization, we simply swap L2 norm with L1 norm. Careful though, the outcome could be very different!

L1 regularization results in a solution that is more sparse. Sparsity in this context refers to the fact that some of the weights end up having an optimal value of zero. Remember the diamond shape of the optimal area?

This property of L1 regularization is extensively used as a feature selection mechanism.

Feature selection simplifies the ML problem by causing a subset of the weights to become zero. Zero weights then highlight the subset of features that can be safely discarded.