Chapter 9: Deep Neural Networks

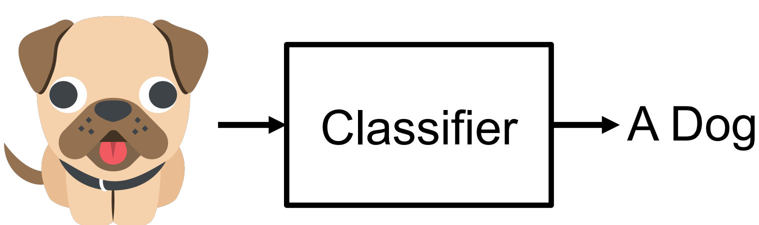
EE2405

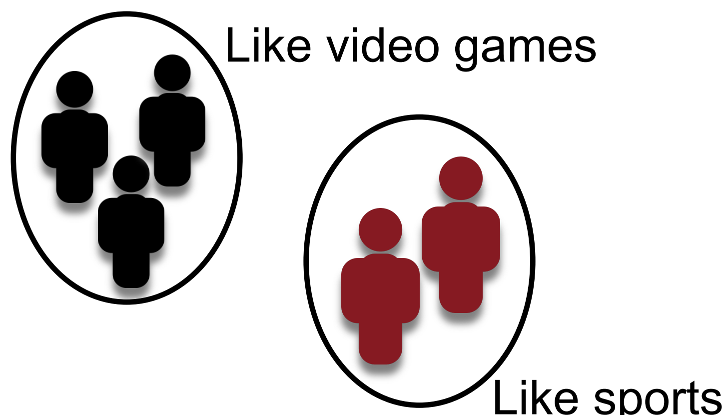
嵌入式系統與實驗

Embedded System Lab

Machine Learning Overview

- Supervised: Learning with a labeled training set
  - Example: classify animal images
- Unsupervised: Discover patterns in unlabeled data
  - Example: cluster people with similar buying habits

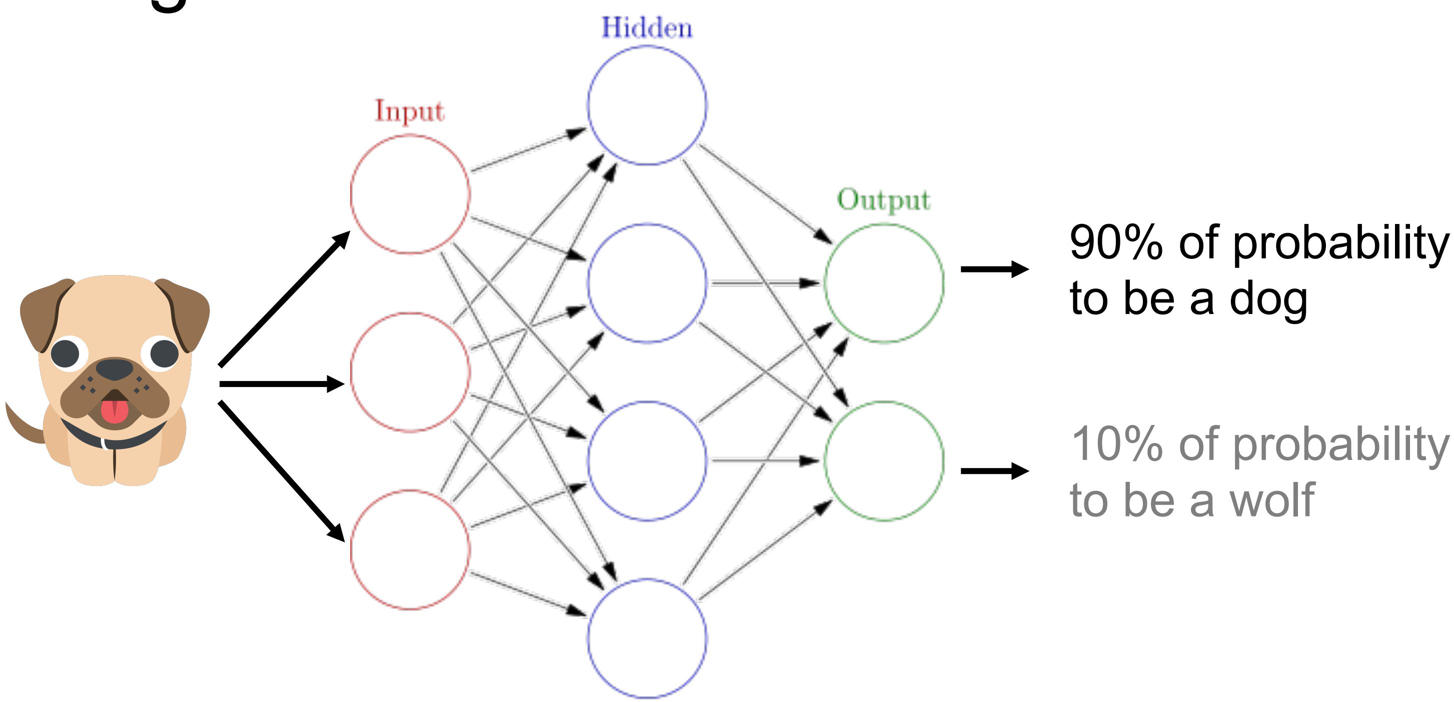




Clustering

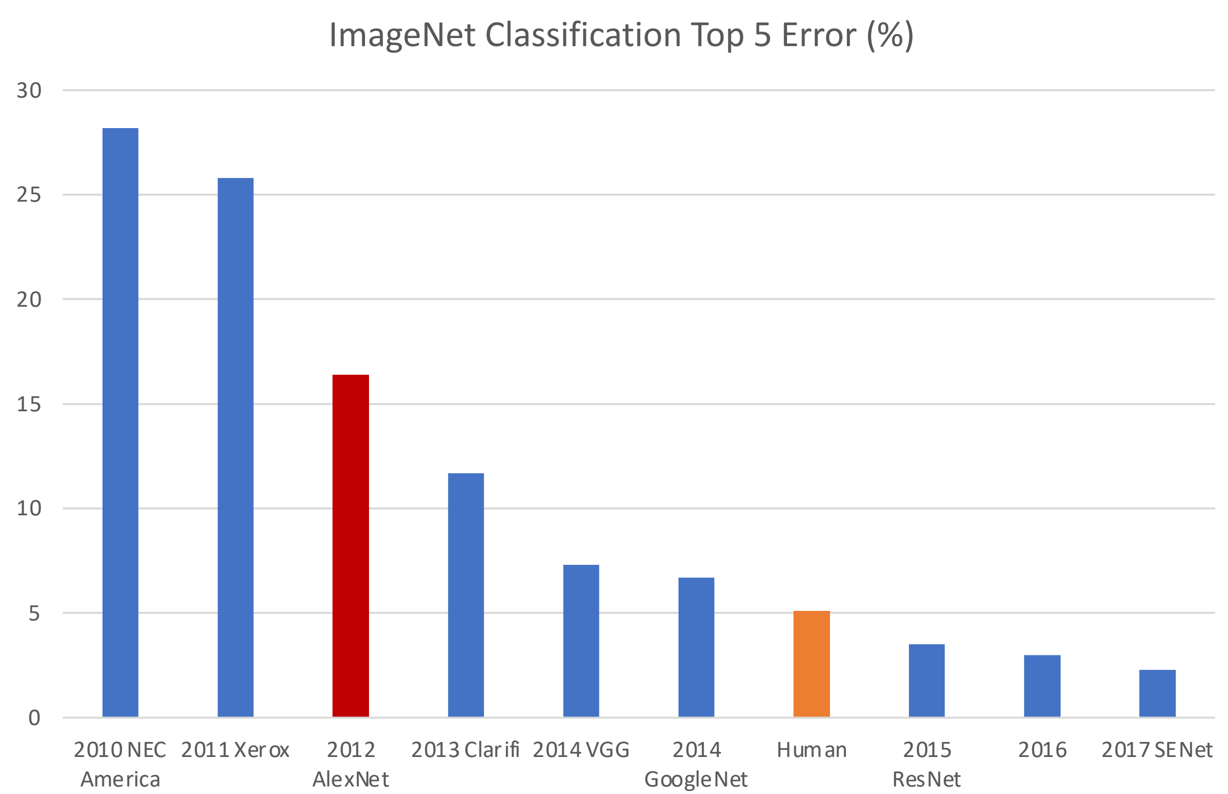
Deep Learning

- One of machine learning techniques
- Apply multi-layer neural networks to machine learning



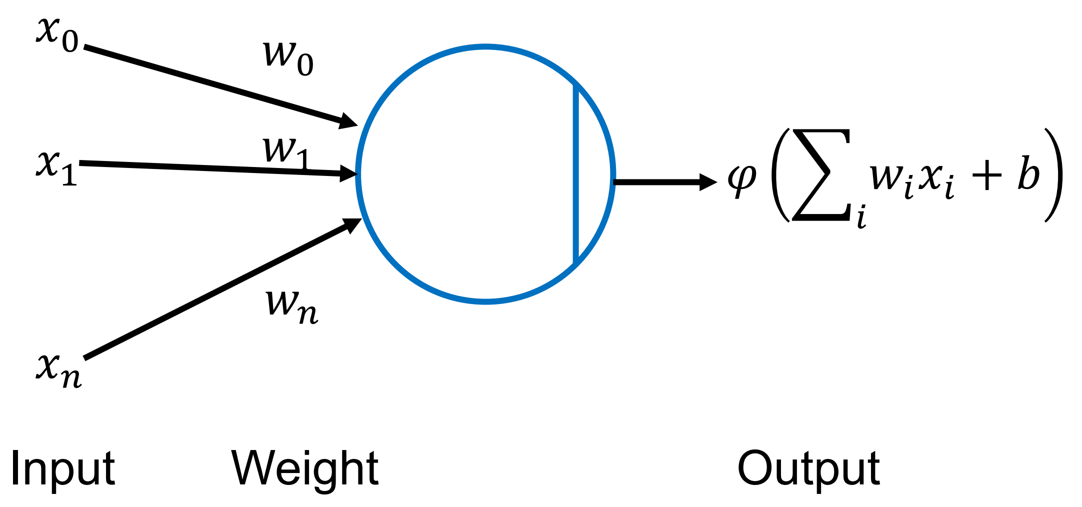
ImageNet Competition

- Convolutional neural networks improve image classification dramatically



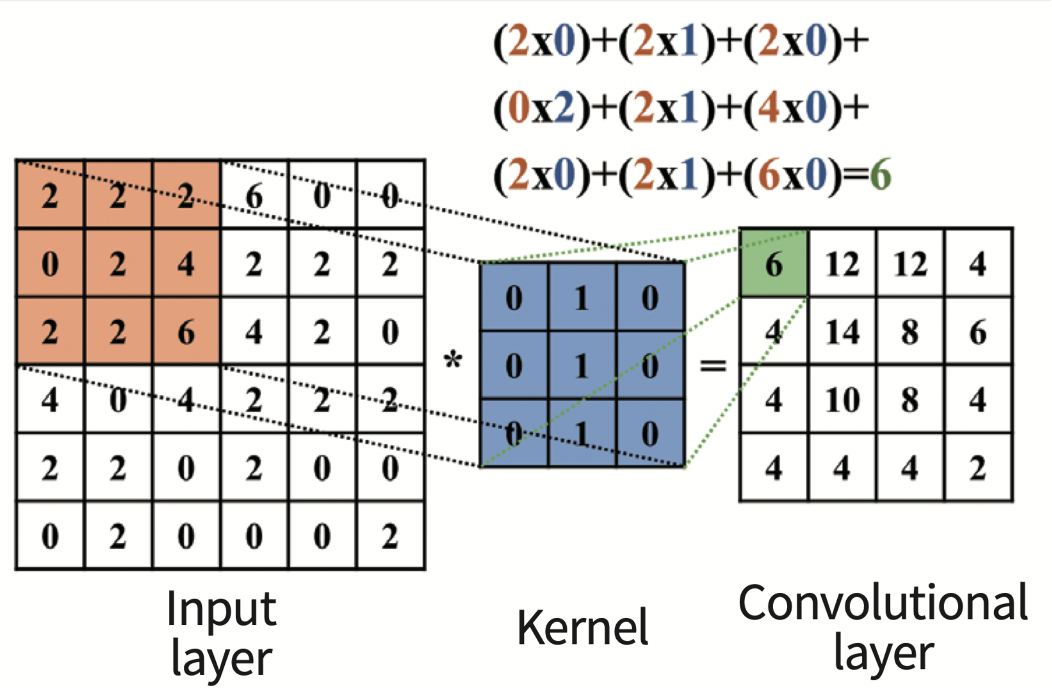
Neuron Model

- A neuron computes weighted sum with bias and pass through an activation function to output



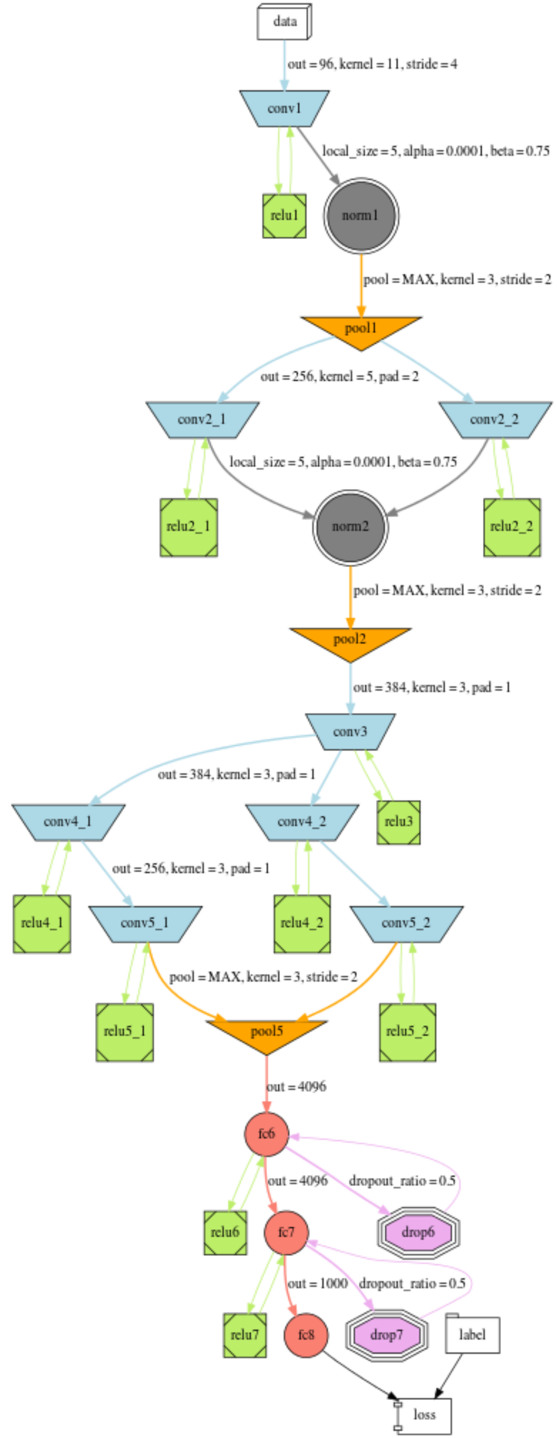
Convolutional Neural Networks

- CNN computes multi-dimension convolution instead of 1-D weighted sums
- For image processing 2-D filters are common



AlexNet

- Convolution
- Relu
- Normalization
- Pooling
- Fully-connected (FC)



Training a Network

- The goal is to optimize (reduce) output loss
  - Need several rounds (epoc)
- Output loss is the difference between known output (100% a dog) and actual prediction (90% a dog)

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights

Repeat till convergence

Discussion on CNN Training

- Prepare labeled (tagged) data
  - For both training and testing
- Use mini-batch
- Avoid overfitting (use dropouts)
- Hyperparameters

Training Frameworks

- In the beginning, Numpy…
- Need more computation resources. Frameworks start to code with GPU: Cafe, Tensorflow, MxNet etc.
  - For specialized networks: Yolo
- To reduce programming efforts: Keras, PyTorch, Tensorflow 2.0, etc.
  - All with Python interface.

A Training Example

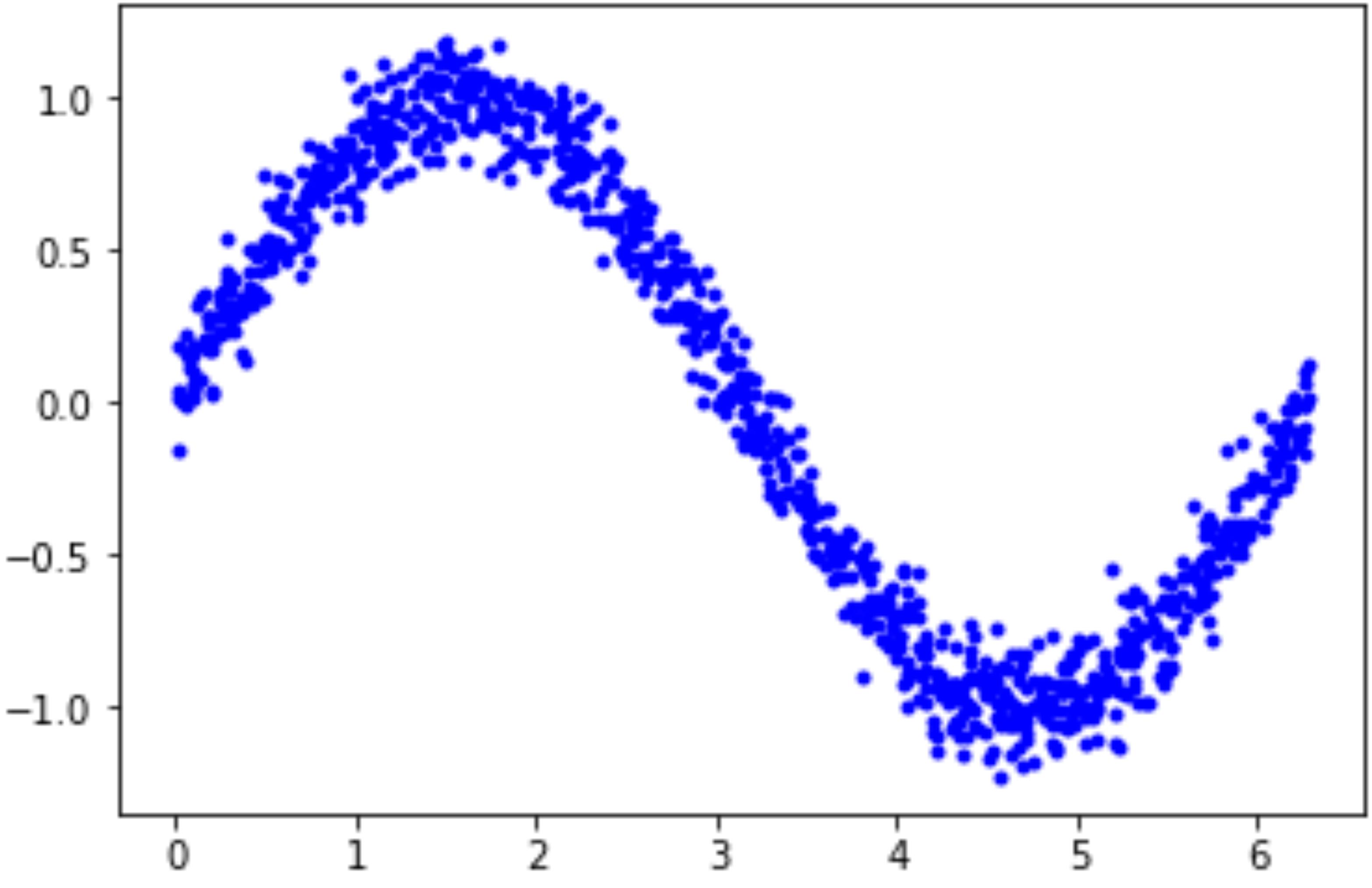
- To train a network to model data generated by a sine function.
  - Data from a sine + some noise with a fix amp/freq
- The example source:
  - https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/hello_world/create_sine_model.ipynb

Training Steps

- Generate data with noise
- Split data into [training, validation, testing]
- Design a model
- Train the model
- Revise the model
- Retrain

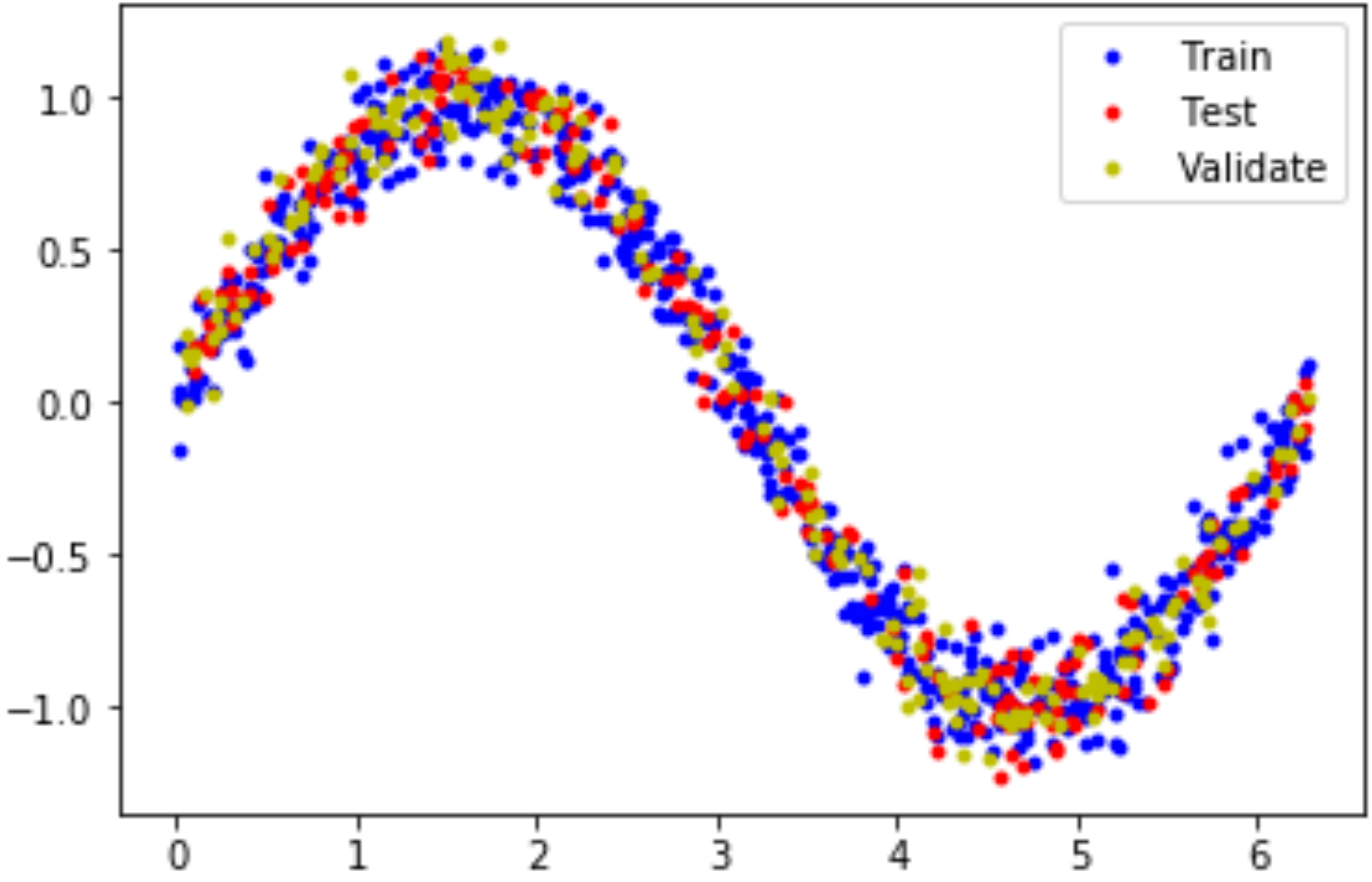
1. Generate data with noise

```
SAMPLES = 1000

np.random.seed(1337)

# Generate a uniformly distributed set of random numbers in [0 to 2π)
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)

# Calculate the corresponding sine values
y_values = np.sin(x_values)
# *operator unpack a list as individual parameters
# randn() to generate data of std normal distribution in y_values.shape
y_values += 0.1 * np.random.randn(*y_values.shape)

# The 'b.' argument tells the library to print blue dots.
plt.plot(x_values, y_values, 'b.')
plt.show()
```


1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Split Data

```
# 60% for training and 20% for testing, 20% for validation.
TRAIN_SPLIT =  int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)

# Use np.split to chop our data into three parts.
# Two indices in 2nd arg, so the data will be divided into three chunks.
x_train, x_test, x_validate = np.split(x_values, [TRAIN_SPLIT,
TEST_SPLIT])
y_train, y_test, y_validate = np.split(y_values, [TRAIN_SPLIT,
TEST_SPLIT])

# Double check that our splits add up correctly
assert (x_train.size + x_validate.size + x_test.size) ==  SAMPLES

# Plot the data in each partition in different colors:
plt.plot(x_train, y_train, 'b.', label="Train")
plt.plot(x_test, y_test, 'r.', label="Test")
plt.plot(x_validate, y_validate, 'y.', label="Validate")
plt.legend()
plt.show()
```



Train
Test
Validate
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Design a Model

```
# TF 2.0 Keras to create a simple model architecture
# Sequential() is a layer by layer model
from tensorflow.keras import layers
model_1 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons". The
# neurons decide whether to activate based on the 'relu' activation
function.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Final layer is a single neuron, since we want to output a single value
model_1.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for
regression
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

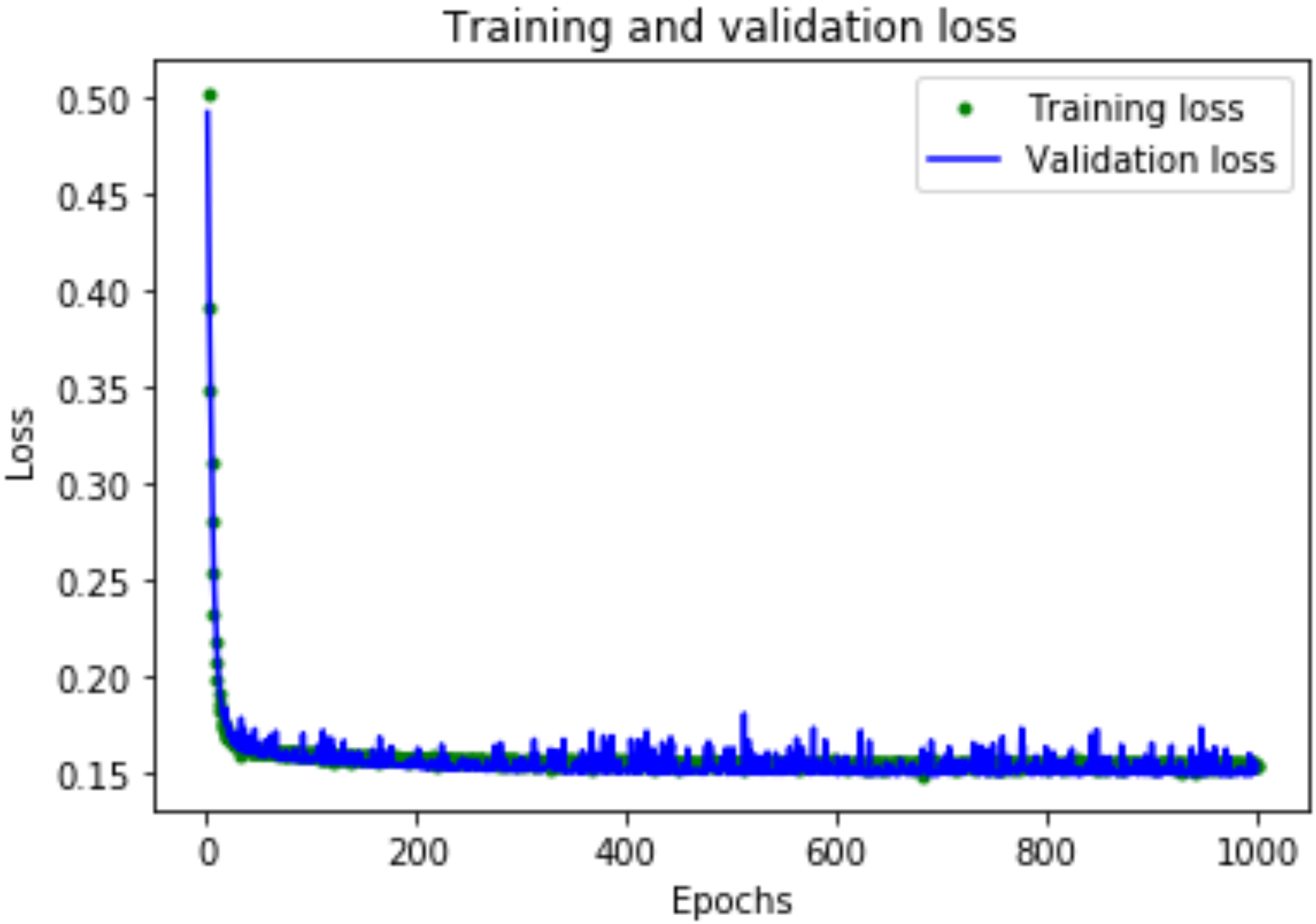
Train the Model

```
# Train the model on our training data while validating on our validation set
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16, validation_data=(x_validate, y_validate))
```

```
Train on 600 samples, validate on 200 samples
Epoch 1/1000
600/600 [==============================] - 0s 412us/sample - loss: 0.5016 - mae: 0.6297 - val_loss: 0.4922 - val_mae: 0.6235
Epoch 2/1000
600/600 [==============================] - 0s 105us/sample - loss: 0.3905 - mae: 0.5436 - val_loss: 0.4262 - val_mae: 0.5641
...
Epoch 998/1000
600/600 [==============================] - 0s 109us/sample - loss: 0.1535 - mae: 0.3068 - val_loss: 0.1507 - val_mae: 0.3113
Epoch 999/1000
600/600 [==============================] - 0s 100us/sample - loss: 0.1545 - mae: 0.3077 - val_loss: 0.1499 - val_mae: 0.3103
Epoch 1000/1000
600/600 [==============================] - 0s 132us/sample - loss: 0.1530 - mae: 0.3045 - val_loss: 0.1542 - val_mae: 0.3143
```

Plot Training and Validation Loss

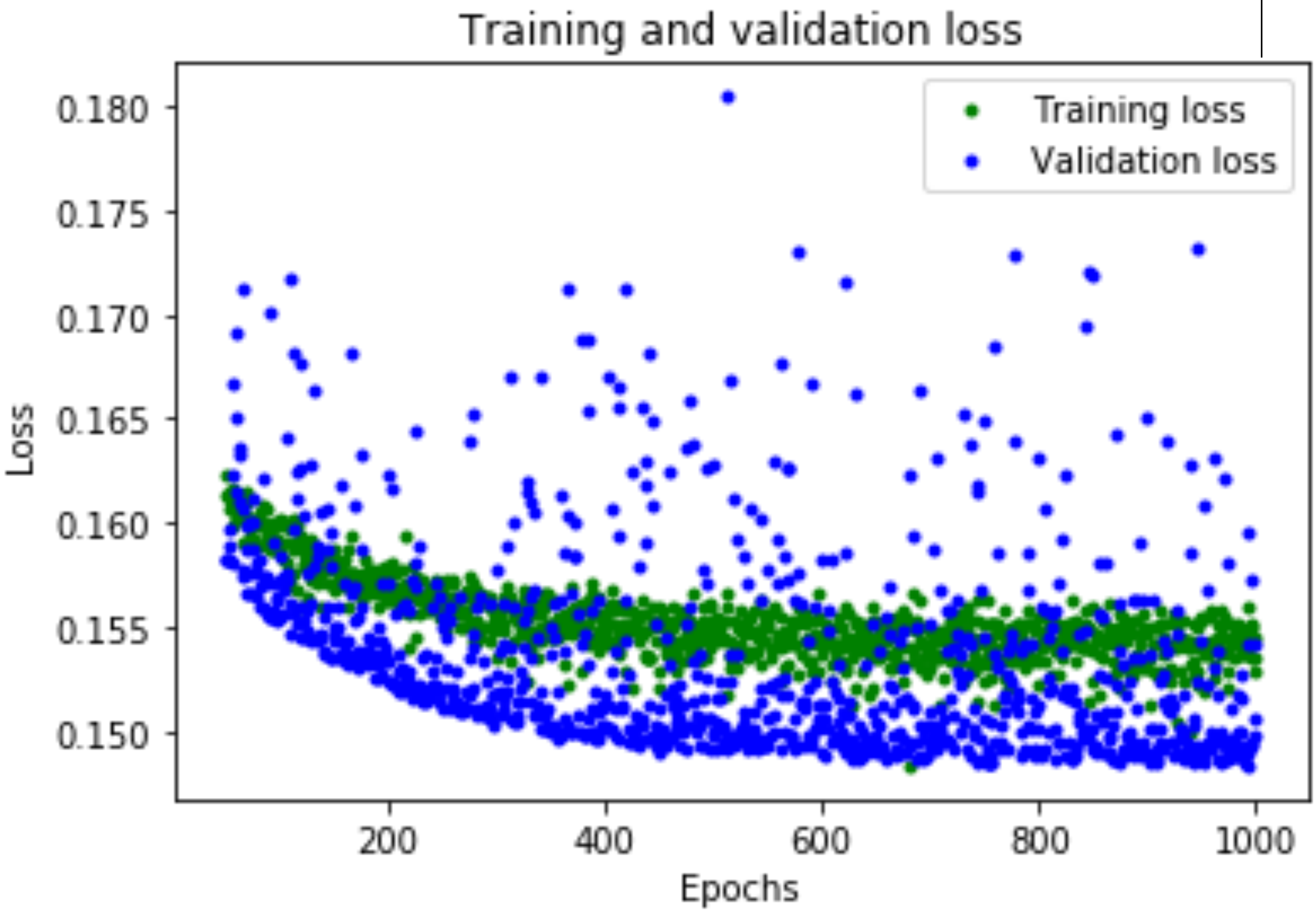
```
# Draw a graph of the loss, which is the distance between
# the predicted and actual values during training and
validation.
loss = history_1.history['loss']
val_loss = history_1.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'g.', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Training and validation loss
Training loss
Validation loss
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
Loss
0
200
400
600
800
1000
Epochs

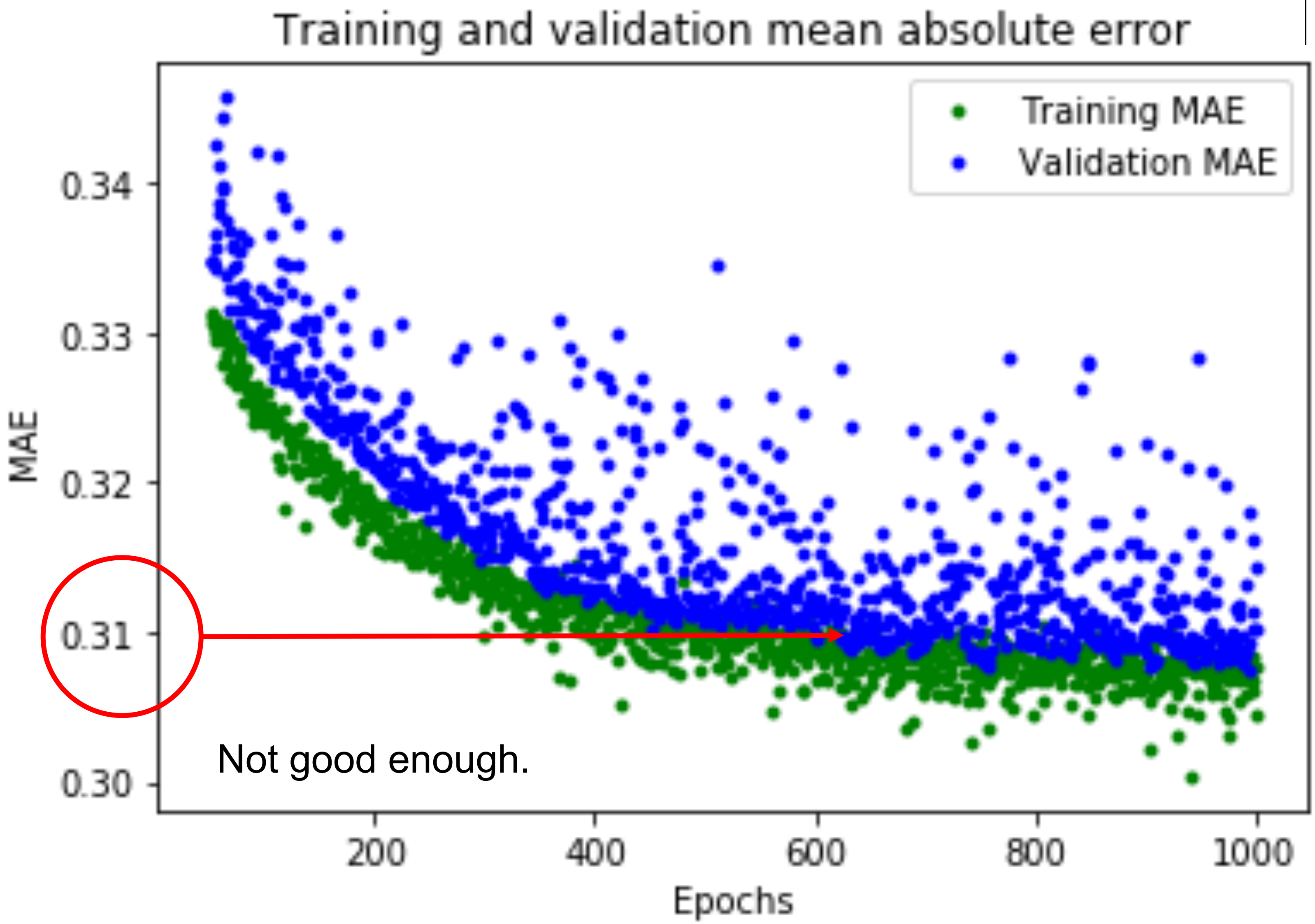
Skip 50 Epocs

```
# Exclude the first few epochs so the graph is
easier to read
SKIP = 50

plt.plot(epochs[SKIP:], loss[SKIP:], 'g.',
label='Training loss')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.',
label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


Training and validation loss
Training loss
Validation loss
Loss
Epochs
0.180
0.175
0.170
0.165
0.160
0.155
0.150
200
400
600
800
1000

Plot Mean Average Error

```
# Draw a graph of mean absolute error, which is another way of
# measuring the amount of error in the prediction.
mae = history_1.history['mae']
val_mae = history_1.history['val_mae']

plt.plot(epochs[SKIP:], mae[SKIP:], 'g.', label='Training MAE')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='Validation MAE')
plt.title('Training and validation mean absolute error')
plt.xlabel('Epochs')
plt.ylabel('MAE')
plt.legend()
plt.show()
```


Training and validation mean absolute error
Training MAE
Validation MAE
0.34
0.33
0.32
0.31
0.30
MAE
200
400
600
800
1000
Epochs
Not good enough.

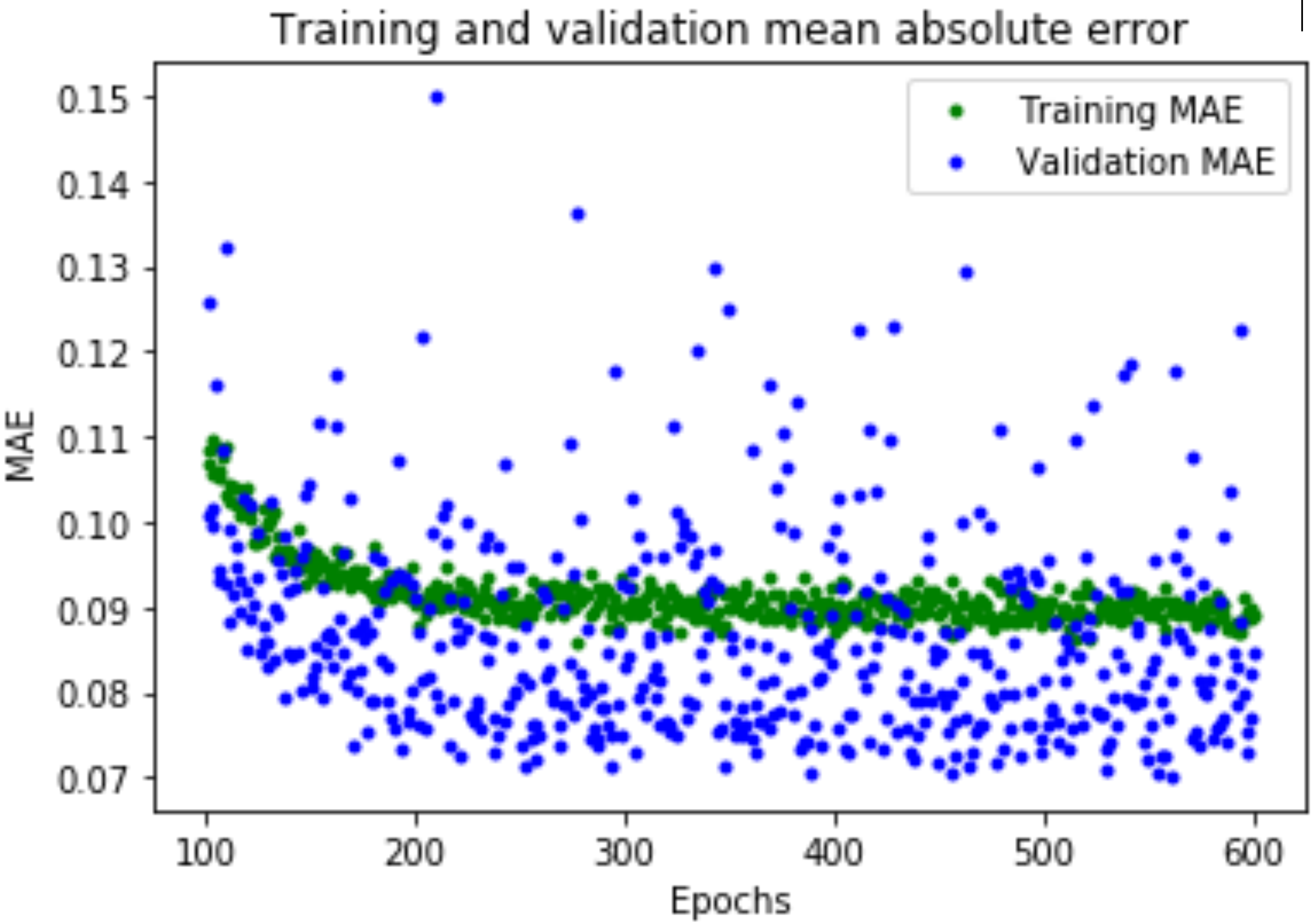
Print Predicted Values (Inference on Training Data)

```
# Use the model to make predictions from our validation data
predictions = model_1.predict(x_train)

# Plot the predictions along with to the test data
plt.clf()
plt.title('Training data predicted vs actual values')
plt.plot(x_test, y_test, 'b.', label='Actual')
plt.plot(x_train, predictions, 'r.', label='Predicted')
plt.legend()
plt.show()
```

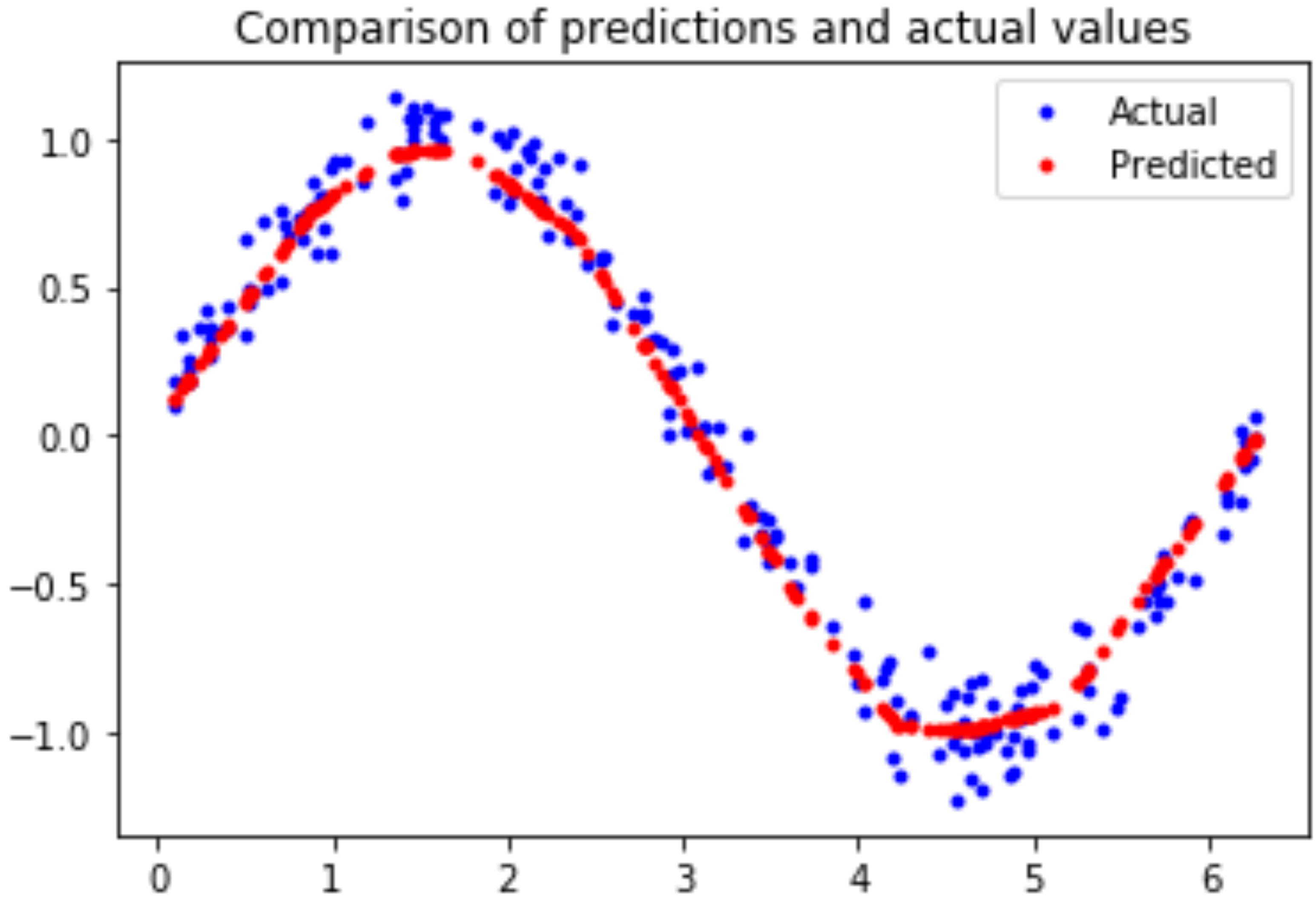
Training data predicted vs actual values
Actual
Predicted
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Add another Layer to the Model

```
model_2 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons". The
# neurons decide whether to activate based on the 'relu' activation
function.
model_2.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# The new second layer may help the network learn more complex
representations
model_2.add(layers.Dense(16, activation='relu'))

# Final layer is a single neuron, since we want to output a single value
model_2.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for
regression
model_2.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

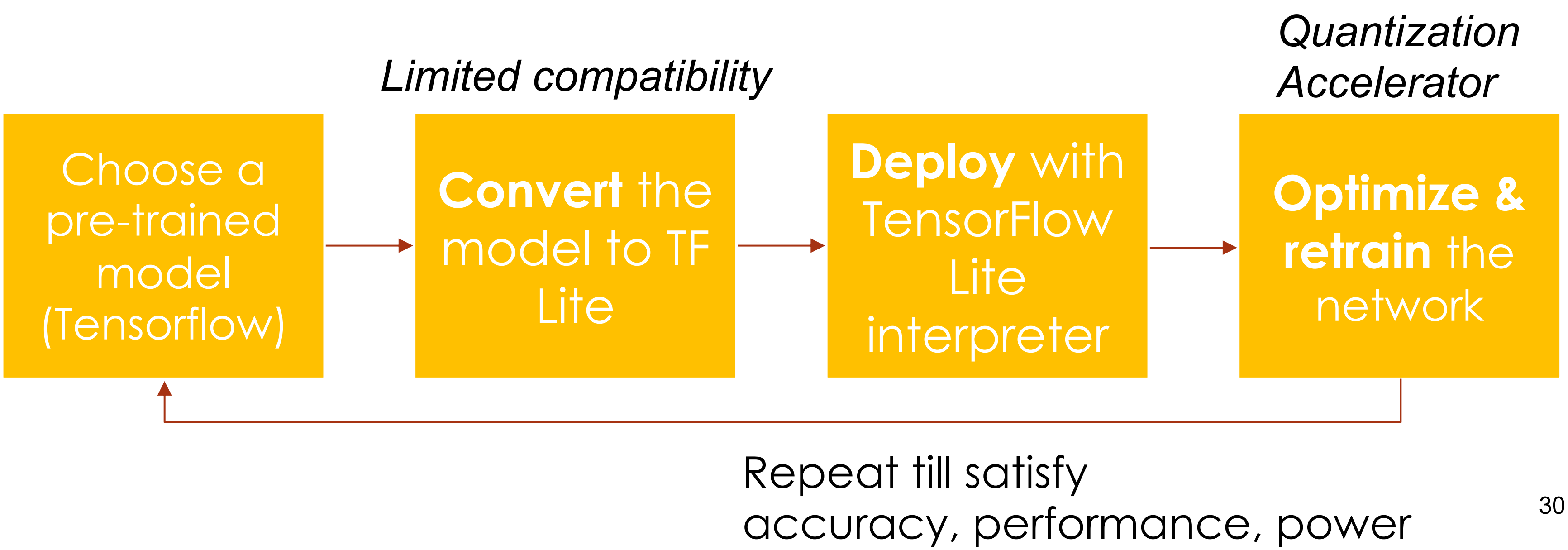


Training and validation mean absolute error
Training MAE
Validation MAE
MAE
Epochs
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
100
200
300
400
500
600


Comparison of predictions and actual values
Actual
Predicted
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Tensorflow Lite

- TensorFlow Lite is an open source deep learning framework for on-device **inference**.
  - Android and IOS devices
  - Micro-controllers



Tensorflow Lite on Micro-controller

- https://www.tensorflow.org/lite/microcontrollers/get_started

Convert to TensorFlow Lite Model

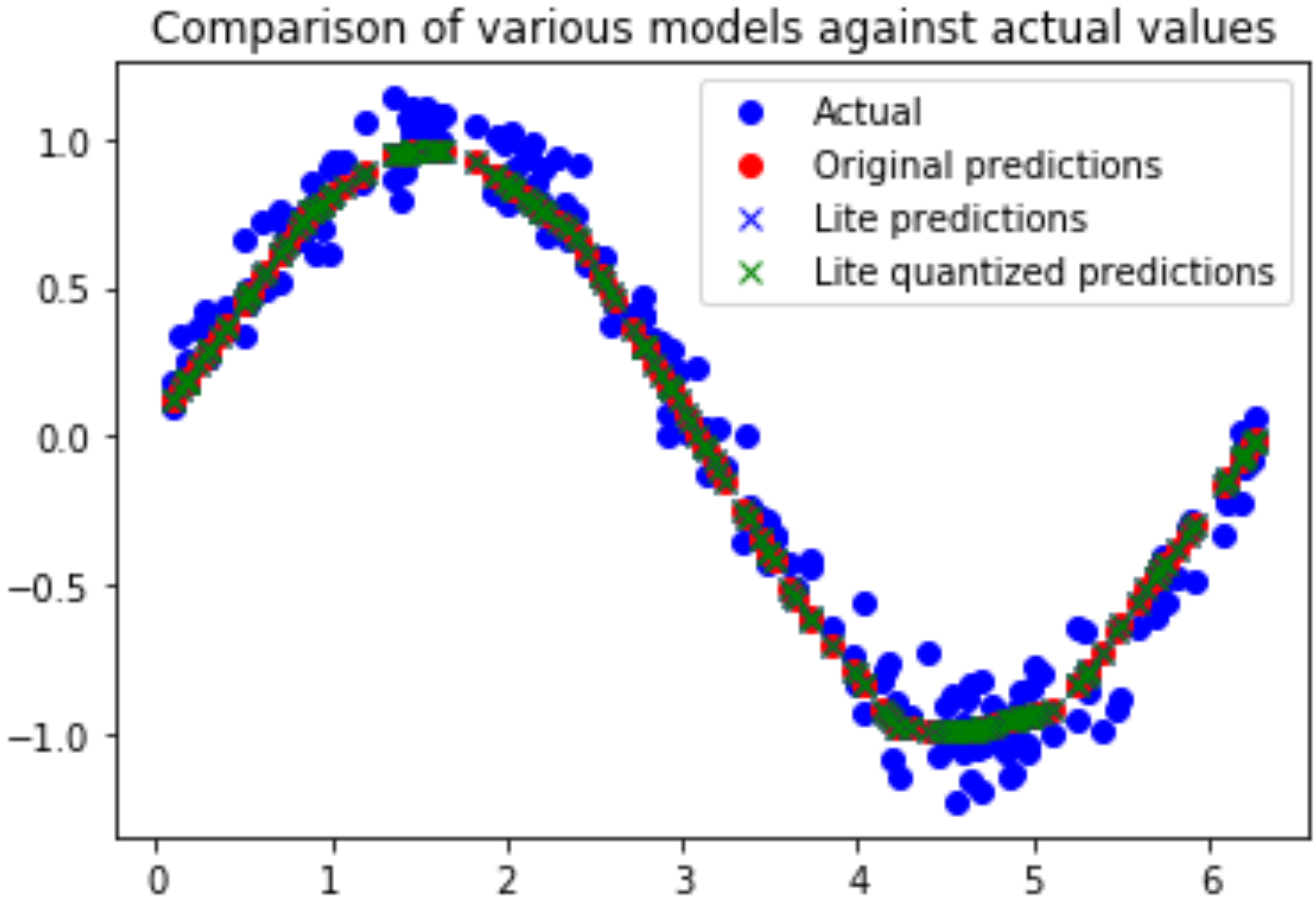
```
# Convert the model to the TensorFlow Lite format without quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
tflite_model = converter.convert()

# Save the model to disk
open("sine_model.tflite", "wb").write(tflite_model)
```

Convert to TensorFlow Lite Quantized Model

```
# Convert the model to the TensorFlow Lite format with quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_model = converter.convert()

# Save the model to disk
open("sine_model_quantized.tflite", "wb").write(tflite_model)
```

Inference in Tensorflow Lite

```
# Instantiate an interpreter for each model
sine_model = tf.lite.Interpreter('sine_model.tflite')

# Allocate memory for each model
sine_model.allocate_tensors()

# Get the input and output tensors so we can feed in values and get the
results
sine_model_input =
sine_model.tensor(sine_model.get_input_details()[0]["index"])
sine_model_output =
sine_model.tensor(sine_model.get_output_details()[0]["index"])

# Create arrays to store the results
sine_model_predictions = np.empty(x_test.size)

# Run each model's interpreter for each value and store the results in
arrays
for i in range(x_test.size):
  sine_model_input().fill(x_test[i])
  sine_model.invoke()
  sine_model_predictions[i] = sine_model_output()[0]
```


Comparison of various models against actual values
Actual
Original predictions
Lite predictions
Lite quantized predictions
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Write to a C File

- Install xxd

$ apt-get -qq install xxd

- Use xxd to convert tflite file to a C hex file

$ xxd -i sine_model_quantized.tflite > sine_model_quantized.cc

```
unsigned char sine_model_quantized_tflite[] = {
0x18, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x0e, 0x00,
0x18, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
…
```

Inference on Micro-controller (1) Includes

```
// sine_model_data.h generated from TF Lite model
// all_ops_resolver.h provides the operations used by the interpreter to run the model.
// micro_error_reporter.h outputs debug information.
// micro_interpreter.h contains code to load and run models.
// schema_generated.h contains the schema for the TensorFlow Lite FlatBuffer model file format.
// version.h provides versioning information for the TensorFlow Lite schema.

#include "tensorflow/lite/micro/examples/hello_world/sine_model_data.h"
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/testing/micro_test.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

Set up Logging

- Setup the error/output mechanism for each micro-controller.

```
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter =
&micro_error_reporter;
```

Load the Model

```
// Map the model into a usable data structure.
// This doesn't involve any copying or parsing,
// it's a very lightweight operation.
const tflite::Model* model =
::tflite::GetModel(g_sine_model_data);

if (model->version() != TFLITE_SCHEMA_VERSION) {
    TF_LITE_REPORT_ERROR(error_reporter,
    "Model provided is schema version %d not equal "
    "to supported version %d.\n",
    model->version(), TFLITE_SCHEMA_VERSION);
}
```

Instantiate Operations Resolver

- To call operators in TF Lite, we need to create a resolver object.

```
// This pulls in all the operation implementations in TF Lite
tflite::ops::micro::AllOpsResolver resolver;
```

- To save memory, and only pull in necessary operators, use `MiroMutableOpResolver`

Allocate Tensor Memories

```
// Create an area of memory to use for
input, output, and intermediate arrays.
// Finding the minimum value for your
model may require some trial and error.
const int tensor_arena_size = 2 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

Instantiate Interpreter

```
// Build an interpreter to run the model
tflite::MicroInterpreter interpreter(model,
resolver, tensor_arena, tensor_arena_size,
error_reporter);
```

Allocate Tensors

- Ask the interpreter to allocate memory from the tensor_arena for the model's tensors

```
interpreter.AllocateTensors();
```

- The `MicroInterpreter` instance can provide us with a pointer to the model's input tensor by calling .input(0), where 0 represents the first (and only) input tensor:

```
// Obtain a pointer to the model's input tensor
TfLiteTensor* input = interpreter.input(0);
```

Validate Input Shape

- Use TF Lite unit test framework

```
// Make sure the input has the properties we expect
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
// The property "dims" tells us the tensor's shape. It has one element for
// each dimension. Our input is a 2D tensor containing 1 element,
// so "dims" should have size 2.
TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);
// The value of each element gives the length of the corresponding tensor.
// We should expect two single element tensors
// (one is contained within the other).
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
// The input is a 32 bit floating point value
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);
```

Run Inference and Get output

```
// Provide an input value
input->data.f[0] = 0.;

// Run the model on this input and check that it succeeds
TfLiteStatus invoke_status = interpreter.Invoke();
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);

// Obtain a pointer to the output tensor
TfLiteTensor* output = interpreter.output(0);

// Obtain the output value from the tensor
float value = output->data.f[0];
// Check that the output value is within 0.05 of
// the expected value
TF_LITE_MICRO_EXPECT_NEAR(0., value, 0.05);
```