Design and Implementation of Deep Neural Networks

Wintersemester 2019/20

Salim Ullah

Chair for Processor Design,
Fakultät Informatik Technische Universität Dresden
1 Introduction

Deep learning (DL) is a subfield of machine learning that is a set of algorithms that is inspired by the structure and function of the brain. The design of a deep neural network (DNN) requires an in-depth understanding of the problem, analyzing application requirements and resource limitations. Based on the analysis, a DNN model is generated, trained, validated, and reiterated. There exist a variety of DNN frameworks for building, training, evaluating, and optimizing a DNN. Fig. 1 summarizes the utilization of state-of-the-art DNN frameworks. These scores are calculated by combining usage, search volume, related publications, and GitHub activity. TensorFlow is the second machine learning framework that Google created and used to design, build, and train deep learning models. It was built to run on multiple CPUs or GPUs and even mobile operating systems, and it has several wrappers in several languages like Python, C++ or Java. It is the most popular deep learning framework today. Gmail, Uber, Airbnb, Nvidia, and lots of other prominent brands are using it. We will also be using TensorFlow in these exercises for defining, training, and evaluating DNNs. These exercises are mainly based on the tutorials provided by https://www.tensorflow.org/tutorials, https://www.easy-tensorflow.com/tf-tutorials/basics/graph-and-session, https://www.guru99.com/what-is-tensorflow.html, https://www.datacamp.com/community/tutorials/tensorflow-tutorial

![Deep Learning Framework Power Scores 2018](image)

Figure 1: Deep Learning Framework Power Scores 2018

1.1 Installation of TensorFlow

TensorFlow is tested and supported on the following 64-bit systems:

1. Ubuntu 16.04 or later
2. Windows 7 or later
3. macOS 10.12.6 (Sierra) or later (no GPU support)
4. Raspbian 9.0 or later

Install TensorFlow with Python’s pip package manager as shown in Listing 1. We would be using Python 3.6. It is recommended to create a virtual environment for all the packages.

```python
# Requires the latest pip
pip install --upgrade pip

# Installation of virtualenv
pip install virtualenv
virtualenv -p /usr/bin/python3.6 venv

# activate the environment
source venv/bin/activate

# install matplotlib and numpy
pip install numpy
pip install matplotlib

# install Tensorflow 2.0
pip install tensorflow

# Or preview build for CPU/GPU (unstable)
pip install tf-nightly
```

Listing 1: Installation of TensorFlow using pip

TensorFlow separates the definition of computations from their execution. Its architecture works in three parts:

1. Preprocessing the data
2. Building the GRAPH (model), it represents the data flow of the computations
3. Running a SESSION, it executes the operations in the graph

### 1.2 What is a Tensor

TensorFlow programs use a data structure called tensor to represent all the data. Any type of data you plan to use for your model can be stored in Tensors. A Tensor is a multi-dimensional array (0-D tensor: scalar, 1-D tensor: vector, 2-D tensor: matrix, and so on). Similar to NumPy ndarray objects, tensor objects of class tensorflow.Tensor have a data type and a shape. Additionally, Tensors objects can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations (tensorflow.add, tensorflow.matmul, tensorflow.linalg.inv etc.) that consume and produce tensorflow.Tensors. These operations automatically convert native Python types. For example, Listing 2 shows the conversion of Python data types to Tensors and matrix multiplication of two tensors. The expected output of each command is also shown in Listing 2.

```python
import tensorflow as tf
print(tf.add(1, 2))
print(tf.add([1, 2], [3, 4]))
print(tf.square(5))
print(tf.reduce_sum([[1, 2, 3]]))
```

Line No.1 in the listing imports tensorflow. This gives Python access to all of TensorFlow’s classes, methods, and symbols. Using this command, TensorFlow library will be imported under the alias tf so that later we can use it instead of typing the whole term tensorflow each time. The most obvious differences between NumPy arrays and tf.Tensors are: (a)Tensors can be backed by accelerator memory (like GPU, TPU), (2) Tensors are immutable.
Operator overloading is also supported

```python
print(tf.square(2) + tf.square(3))
```

```python
x = tf.matmul([[1]], [[2, 3]])
print(x)
p(x.shape)
p(x.dtype)
```

# Generated Output
```
# tf.Tensor(3, shape=(), dtype=int32)
# tf.Tensor([4 6], shape=(2,), dtype=int32)
# tf.Tensor(25, shape=(), dtype=int32)
# tf.Tensor(13, shape=(), dtype=int32)
# tf.Tensor([[2 3]], shape=(1, 2), dtype=int32)
(1, 2)
<dtype: 'int32'>
```

Listing 2: Examples of TensorFlow commands

### 1.3 Computational Graph

A computational graph (or graph in short) is a series of TensorFlow operations arranged into a graph of nodes. It means a graph is just an arrangement of nodes that represent the operations in your model. For example, for the function \( f(x, y) = x^2y + y + 2 \), the computational graph generated by TensorFlow would be something like as shown in Fig. 2. The graph is composed of a series of nodes connected to each other by edges. Each node in the graph is called op (short for operation). So there is one node for each operation; either for operations on tensors (like math operations) or generating tensors (like variables and constants). Each node takes zero or more tensors as inputs and produces a tensor as an output.

Example 1.1. Let’s start with a basic arithmetic operation like addition to demonstrate a graph. The code adds two values, say \( a=2 \) and \( b=3 \), using TensorFlow. To do so, we need to call `tf.add()`. The `tf.add()` has three arguments ‘x’, ‘y’, and ‘name’ where \( x \) and \( y \) are the values to be added together and name is the operation name, i.e. the name associated to the addition node on the graph. Listing 3 describes the example code for this operation. This code creates two input nodes (for inputs \( a=2 \) and \( b=3 \)) and one output node for the addition operation. Each operation in TensorFlow can be assigned an optional name, as shown by the name=‘Add’ segment. The output ‘c’ is a tensor of the same data type as the input tensors to the `tf.add` operation. When we print out the variable c (i.e. the output Tensor of the addition operation), it prints out the Tensor information; its name (Add), shape (1 means scalar), and type (32-bit integer). However, it does not print out the result (2+3=5). This behavior is due to the reason that lines No. 2–5 only defines

![Figure 2: Schematic of the constructed computational graph in TensorFlow](image-url)
the computational graph. The sample computational graph is shown in Fig 3. To evaluate the nodes, we must run the computational graph either with a Function call (TensorFlow 2 only) or within a Session (TensorFlow 1.X only). The written code only generates the graph, which only determines the expected sizes of Tensors and operations to be executed on them. However, it does not assign a numeric value to any of the Tensors i.e., TensorFlow does not execute the graph unless it is specified to do so with a function/session.

**Session: Effective for TensorFlow 1.X**

To compute anything, a graph must be launched in a session. Technically, session places the graph ops on hardware such as CPUs or GPUs and provides methods to execute them. In our example, to run the graph and get the value for ‘c’ the code in Listing 4 will create a session and execute the graph by running ‘c’. This code creates a Session object (assigned to sess), and then (the second line) invokes its run method to execute enough of the computational graph to evaluate output ‘c’. This means that it only runs that part of the graph which is necessary to get the value of c. In this example, it runs the whole graph. Remember to close the session at the end of the session. That is done using the last line in the above code. The code in Listing 5 does the same thing and is more commonly used. The only difference is that there is no need to close the session at the end as it gets closed automatically.

```python
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
print(c)
```

Listing 3: Addition of two tensors

```python
# Generated Output
# Tensor("Add:0", shape=(), dtype=int32)
```

Graph

![Graph](image)

Figure 3: Generated graph for Listing 3 visualized in Tensorboard

```python
sess = tf.Session()
print(sess.run(c))
sess.close()
```

Listing 4: Session to run graph defined in Listing 3

```python
# Generated Output
# 5
```

```python
with tf.Session() as sess:
    print(sess.run(c))
```

Listing 5: Session to run graph defined in Listing 3

**Function call: Effective for TensorFlow 2**

A session.run() call is almost like a function call: You specify the inputs and the function to be called, and you get back a set of outputs. In TensorFlow 2.0, you can decorate a Python function using tf.function() to mark it for JIT compilation so that TensorFlow runs it as a single graph (Functions 2.0 RFC). This mechanism allows TensorFlow 2.0 to gain all of the benefits of graph mode.
import tensorflow as tf
a = 2
b = 3
@tf.function
def f():
c = tf.add(a, b, name='Add')
return c
print(f())
print(f().numpy())

Listing 6: Graph generation and execution using TensorFlow 2 for the graph defined in Listing 3

Example 1.2. Consider the code in Listing 7 and corresponding generated computational graph in Fig. 4. Given this graph, if we fetch the pow_op operation, it will first run the add_op and mul_op to get their output tensor and then run pow_op on them to compute the required output value. In other words, useless_op will not be executed as it’s output tensor is not used in executing the pow_op operation. This specially saves a significant amount of time for us when dealing with huge networks with hundreds and thousands of operations.

Listing 7: Computational graph generation and execution for different operations using TensorFlow 1.X

import tensorflow as tf
x = 2
y = 3
add_op = tf.add(x, y, name='Add')
mul_op = tf.multiply(x, y, name='Multiply')
pow_op = tf.pow(add_op, mul_op, name='Power')
useless_op = tf.multiply(x, add_op, name='Useless')
with tf.Session() as sess:
pow_out, useless_out = sess.run([pow_op, useless_op])

Listing 8 shows the implementation of the graph in Fig. 4 in TensorFlow 2. A common usage pattern in TensorFlow 1.X was the “kitchen sink” strategy, where the union of all possible computations was preemptively laid out, and then selected tensors were evaluated via session.run(). In TensorFlow 2.0, users should refactor their code into smaller functions which are called as needed. In general, it’s not necessary to decorate each of these smaller functions with tf.function; only use tf.function to decorate high-level computations.
@tf.function

def f():
    add_op = tf.add(x, y, name='Add')
    mul_op = tf.multiply(x, y, name='Multiply')
    pow_op = tf.pow(add_op, mul_op, name='Power')
    useless_op = tf.multiply(x, add_op, name='Useless')
    return pow_op, useless_op

print(f())

Listing 8: Computational graph generation and execution for different operations using TensorFlow 2

Example 1.3. Fit a linear model

You will create a simple linear model, \( f(x) = x \ast W + b \), which has two variables: \( W \) (weights) and \( b \) (bias). You will synthesize data such that a well trained model would have \( W = 3.0 \) and \( b = 2.0 \). The following concepts would be used for building the model:

- **Variables:** Use tf.Variable to represent weights in a model. A tf.Variable object stores a value and implicitly reads from this stored value. There are operations (tf.assign_sub, tf.scatter_update, etc.) that manipulate the value stored in a TensorFlow variable. Trainable variables (created by tf.Variable or tf.compat.v1.get_variable, where trainable=True is default in both cases) are automatically watched. Tensors can be manually watched by invoking the watch method on this context manager.

- **Gradient tapes:** TensorFlow provides the tf.GradientTape API for automatic differentiation - computing the gradient of a computation with respect to its input variables. Listing 9 provides an example of automatic differentiation using gradient tapes.

```
import tensorflow as tf
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x
dy_dx = g.gradient(y, x) # Will compute to 6.0
```

Listing 9: Gradient tape example

Building and training a model consists of the following steps:

- Define the model.
- Define a loss function.
- Obtain training data.
- Run through the training data and use an "optimizer" to adjust the variables to fit the data.

**Define the model:** Let’s define a simple class to encapsulate the variables and the computation. Consider Listing 10:

```
import tensorflow as tf
import matplotlib.pyplot as plt
class Model(object):
    def __init__(self):
        # Initialize the weights to '5.0' and the bias to '0.0'
        # In practice, these should be initialized to random values (for example, with 'tf.random.normal')
```
```python
self.W = tf.Variable(5.0)
self.b = tf.Variable(0.0)
def __call__(self, x):
    return self.W * x + self.b

model = Model()

assert model(3.0).numpy() == 15.0
```

Listing 10: Model definition for linear model

Define a loss function: A loss function measures how well the output of a model for a given input matches the target output. The goal is to minimize this difference during training. Let’s use the standard L2 loss, also known as the least square errors. Consider Listing 11:

```python
def loss(predicted_y, target_y):
    return tf.reduce_mean(tf.square(predicted_y - target_y))
```

Listing 11: Loss function for linear model

Obtain training data: First, synthesize the training data by adding random Gaussian (Normal) noise to the inputs. The corresponding code is available in Listing 12:

```python
TRUE_W = 3.0
TRUE_b = 2.0
NUM_EXAMPLES = 1000
inputs = tf.random.normal(shape=[NUM_EXAMPLES])
noise = tf.random.normal(shape=[NUM_EXAMPLES])
outputs = inputs * TRUE_W + TRUE_b + noise
```

Listing 12: Training data for linear model

Before training the model, visualize the loss value by plotting the model’s predictions in red and the training data in blue. Consider Listing 13 and Fig. 5:

```python
plt.scatter(inputs, outputs, c='b')
plt.scatter(inputs, model(inputs), c='r')
plt.show()

print('Current loss: %1.6f' % loss(model(inputs), outputs).numpy())
```

Listing 13: Pre-Training loss of the model’s prediction

Define a training loop: With the network and training data, train the model using gradient descent to update the weights variable (W) and the bias variable (b) to reduce the loss. There are many variants of the gradient descent scheme that are captured in tf.train.Optimizer. But here you will implement the basic math yourself with the help of tf.GradientTape for automatic differentiation and tf.assign_sub for decrementing a value (which combines tf.assign and tf.sub). Refer to Listing 14, Fig. 6 and Fig. 7:

```python
def train(model, inputs, outputs, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(model(inputs), outputs)
dW, db = t.gradient(current_loss, [model.W, model.b])
model.W.assign_sub(learning_rate * dW)
model.b.assign_sub(learning_rate * db)

# Repeatedly run through the training data
model = Model()

# Collect the history of W-values and b-values to plot later
Ws, bs = [], []
ePOCHS = range(10)
```
for epoch in epochs:
    Ws.append(model.W.numpy())
    bs.append(model.b.numpy())
    current_loss = loss(model(inputs), outputs)

    print('Epoch %2d: W =%1.2f b =%1.2f, loss =%2.5f' %
          (epoch, Ws[-1], bs[-1], current_loss))

# Let's plot it all
plt.plot(epochs, Ws, 'r', epochs, bs, 'b')
plt.plot([TRUE_W] * len(epochs), 'r--', [TRUE_b] * len(epochs), 'b--')
plt.legend(['W', 'b', 'True W', 'True b'])
plt.xlabel('Epochs')
plt.ylabel('Value')
plt.show()

# Plot again the outputs of the trained model
plt.scatter(inputs, outputs, c='b')
plt.scatter(inputs, model(inputs), c='r')
plt.legend(['Training Data', 'Prediction'])
plt.xlabel('Inputs')
plt.ylabel('Outputs')
plt.show()

print('Current loss: %1.6f' % loss(model(inputs), outputs).numpy())

# Generated loss values
# Epoch 0: W=5.00 b=0.00, loss=8.96391
# Epoch 1: W=4.59 b=0.39, loss=6.11565
# Epoch 2: W=4.27 b=0.70, loss=4.28201
# Epoch 3: W=4.01 b=0.95, loss=3.10103
# Epoch 4: W=3.80 b=1.16, loss=2.34008
# Epoch 5: W=3.64 b=1.33, loss=1.84957
# Epoch 6: W=3.51 b=1.48, loss=1.53327
# Epoch 7: W=3.41 b=1.57, loss=1.32923
# Epoch 8: W=3.33 b=1.66, loss=1.19756
# Epoch 9: W=3.26 b=1.73, loss=1.12257
# Current loss: 1.057693

Listing 14: Training loop for training W and b
2 Building and Training an Artificial Neural Network

The Iris classification

A machine learning program could classify flowers based on photographs. We are going to classify Iris flowers based on the length and width measurements of their sepals and petals. The Iris genus entails about 300 species, but our program will only classify the following three:

- Iris setosa
- Iris virginica
- Iris versicolor

We will be using the following steps:

- Import and parse the dataset (using Datasets API).
- Select the type of model.
- Train the model (using Keras API).
- Evaluate the model’s effectiveness.

Configure imports and download the training dataset

A dataset of 120 Iris flowers with the sepal and petal measurements is already available as a CSV file. Listing [15] imports the required packages and downloads the training data. The first line in the CSV file is a header containing information about the dataset:

- There are 120 total examples. Each example has four features and one of three possible label names.
- The first four fields are features: these are the characteristics of an example. Here, the fields hold float numbers representing flower measurements.
- The last column is the label: this is the value we want to predict. For this dataset, it’s an integer value of 0, 1, or 2 that corresponds to a flower name.
Figure 7: Output of the linear model after training

```
from __future__ import absolute_import, division, print_function, unicode_literals

import os
import matplotlib.pyplot as plt
import tensorflow as tf

# Download training data
train_dataset_url = "https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv"

train_dataset_fp = tf.keras.utils.get_file(fname=os.path.basename(train_dataset_url),
origin=train_dataset_url)

print("Local copy of the dataset file: {}").format(train_dataset_fp)

# Create a tf.data.Dataset for the model
TensorFlow’s Dataset API handles many common cases for loading data into a model. Since the dataset is a CSV-formatted text file, use the make_csv_dataset function to parse the data into a suitable format. Each label is associated with string name (for example, “setosa”), but machine learning typically relies on numeric values. The label numbers are mapped to a named representation, such as:

- Iris setosa
- Iris versicolor
- Iris virginica

# column order in CSV file
column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']

feature_names = column_names[:-1]
label_name = column_names[-1]

print("Features: {}".format(feature_names))
print("Label: {}".format(label_name))
class_names = ['Iris setosa', 'Iris versicolor', 'Iris virginica']
```
batch_size = 32

train_dataset = tf.data.experimental.make_csv_dataset(
    train_dataset_fp,
    batch_size,
    column_names=column_names,
    label_name=label_name,
    num_epochs=1)

Listing 16: Converting data from CSV format to Tensorflow dataset format

The `make_csv_dataset` function returns a tf.data.Dataset of (features, label) pairs, where features is a dictionary: `feature_name': value. These Dataset objects are iterable. Let’s look at a batch of features in Listing 17.

features, labels = next(iter(train_dataset))

Listing 17: Printing a sample of dataset

To simplify the model building step, create a function to repackage the features dictionary into a single array with shape: (batch_size, num_features). This function uses the tf.stack method which takes values from a list of tensors and creates a combined tensor at the specified dimension, as described in Listing 18. The features element of the Dataset are now arrays with shape (batch_size, num_features). Listing 18 also shows few examples:

def pack_features_vector(features, labels):
    features = tf.stack(list(features.values()), axis=1)
    return features, labels

train_dataset = train_dataset.map(pack_features_vector)

# Example
features, labels = next(iter(train_dataset))

Listing 18: Printing a sample of dataset

Neural Network Model Selection

We need to select the kind of model to train. Fig. 8 illustrates a dense neural network consisting of an input layer, two hidden layers, and an output layer. When this model is trained and fed an unlabeled example, it yields three predictions: the likelihood that this flower is the given Iris species.

Create a model using Keras

The TensorFlow tf.keras API is the preferred way to create models and layers. The tf.keras.Sequential model is a linear stack of layers. Its constructor takes a list of layer instances, in this case, two Dense layers with 10 nodes each, and an output layer with 3 nodes representing our label predictions. The first layer’s input_shape parameter corresponds to the number of features from the dataset, and is required. The model is defined in Listing 19. The activation function determines the output shape of each node in the layer.

model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,), input_shape # required
    tf.keras.layers.Dense(10, activation=tf.nn.relu),
    tf.keras.layers.Dense(3)
])

Listing 19: Keras sequential model

Using the model

Let’s have a quick look at what this model does to a batch of features. Listing 20 provides a batch of features to the model and prints the output of the last layer (line 1 and 2). Here, each example returns a logit for each
Logits is the vector of raw (non-normalized) predictions that a classification model generates, which is ordinarily then passed to a normalization function. If the model is solving a multi-class classification problem, logits typically become an input to the softmax function (line 6). The softmax function then generates a vector of (normalized) probabilities with one value for each possible class. Taking the \texttt{tf.argmax} (line 9) across classes gives us the predicted class index. But, the model hasn’t been trained yet, so these aren’t good predictions:

```python
predictions = model(features)
print(predictions[:5])

# Conversion of logits to probabilities using softmax

# take maximum of the probabilities to find the identified class
print("Prediction: \{\}".format(tf.argmax(predictions, axis=1)))
print("Labels: \{\}".format(labels))
```

Listing 20: Using the model without training

Train the model

Training is the stage of machine learning when the model is gradually optimized, or the model learns the dataset. The Iris classification problem is an example of supervised machine learning: the model is trained from examples that contain labels.

As we did previously, we would define the loss and gradient functions. Both training and evaluation stages need to calculate the model’s loss. This measures how off a model’s predictions are from the desired label, in other words, how bad the model is performing. We want to minimize, or optimize, this value. Our model will calculate its loss using the \texttt{tf.keras.losses.SparseCategoricalCrossentropy} function which takes the model’s class probability predictions and the desired label, and returns the average loss across the examples. Consider Listing 21

Line 1 defines the loss object. This loss object is used by the custom loss function (lines 3–5). Lines 8 and 9 show an example of using the loss function. Use the \texttt{tf.GradientTape} context to calculate the gradients used to optimize your model(lines 12–15). Finally, an optimizer applies the computed gradients to the model’s variables to minimize the loss function. TensorFlow has many optimization algorithms available for training. This model uses the \texttt{tf.keras.optimizers.SGD} that implements the \textit{stochastic gradient descent (SGD)} algorithm. The learning rate sets the step size to take for each iteration.

```python
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
def loss(model, x, y):
y_ = model(x)
return loss_object(y_true=y, y_pred=y_)
```
# Example of using loss function
l = loss(model, features, labels)
print("Loss test: {}".format(l))

# Gradient definition using tf.GradientTape
def grad(model, inputs, targets):
    with tf.GradientTape() as tape:
        loss_value = loss(model, inputs, targets)
        return loss_value, tape.gradient(loss_value, model.trainable_variables)

# Optimizer for reducing the loss using gradients
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

Listing 21: Defining the loss gradient and optimization functions

Training loop

A training loop feeds the dataset examples into the model to help it make better predictions. The following code block sets up these training steps:

- Iterate each epoch. An epoch is one pass through the dataset.
- Within an epoch, iterate over each example in the training Dataset grabbing its features (x) and label (y).
- Using the example’s features, make a prediction and compare it with the label. Measure the inaccuracy of the prediction and use that to calculate the model’s loss and gradients.
- Use an optimizer to update the model’s variables.
- Keep track of some stats for visualization.
- Repeat for each epoch.

These steps are performed by Listing 22. While it’s helpful to print out the model’s training progress, it’s often more helpful to see this progress. We can create basic charts using the matplotlib module in Listing 23.

## Note: Rerunning this cell uses the same model variables

```python
# Keep results for plotting
train_loss_results = []
train_accuracy_results = []

num_epochs = 201

for epoch in range(num_epochs):
    epoch_loss_avg = tf.keras.metrics.Mean()
    epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

    # Training loop - using batches of 32
    for x, y in train_dataset:
        # Optimize the model
        loss_value, grads = grad(model, x, y)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        # Track progress
        epoch_loss_avg(loss_value)  # Add current batch loss
        epoch_accuracy(y, model(x))

    # End epoch
    train_loss_results.append(epoch_loss_avg.result())
    train_accuracy_results.append(epoch_accuracy.result())

    if epoch % 50 == 0:
        print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch, epoch_loss_avg.result(), epoch_accuracy.result()))
```

Listing 22: Training loop implementation
Listing 22: Training loop

```python
fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))
fig.suptitle('Training Metrics')
axes[0].set_ylabel("Loss", fontsize=14)
axes[0].plot(train_loss_results)
axes[1].set_ylabel("Accuracy", fontsize=14)
axes[1].set_xlabel("Epoch", fontsize=14)
axes[1].plot(train_accuracy_results)
plt.show()
```

Listing 23: Visualize the loss function and accuracy

**Setup the test dataset**

Evaluating the model is similar to training the model. The biggest difference is the examples come from a separate test set rather than the training set. The setup for the test Dataset is similar to the setup for training Dataset. Unlike the training stage, the model only evaluates a single epoch of the test data. In Listing[24] we iterate over each example in the test set and compare the model’s prediction against the actual label. This is used to measure the model’s accuracy across the entire test set:

```python
test_url = "https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv"
test_fp = tf.keras.utils.get_file(fname=os.path.basename(test_url), origin=test_url)
test_dataset = tf.data.experimental.make_csv_dataset(
    test_fp,
    batch_size,
    column_names=column_names,
    label_name='species',
    num_epochs=1,
    shuffle=False)
test_dataset = test_dataset.map(pack_features_vector)
test_accuracy = tf.keras.metrics.Accuracy()
for (x, y) in test_dataset:
    logits = model(x)
    prediction = tf.argmax(logits, axis=1, output_type=tf.int32)
    test_accuracy(prediction, y)
print("Test set accuracy: {:.3%}".format(test_accuracy.result()))
```

Listing 24: Evaluating the trained model

Combining all the pieces together, Listing 25 shows the complete code for data downloading, model development, training and evaluation.

```python
from __future__ import absolute_import, division, print_function, unicode_literals

import os
import matplotlib.pyplot as plt
import tensorflow as tf

train_dataset_url = "https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv"
train_dataset_fp = tf.keras.utils.get_file(fname=os.path.basename(train_dataset_url), origin=train_dataset_url)

# column order in CSV file
column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
```
feature_names = column_names[: -1]
label_name = column_names[-1]
class_names = ['Iris setosa', 'Iris versicolor', 'Iris virginica']
batch_size = 32
train_dataset = tf.data.experimental.make_csv_dataset(
    train_dataset_fp,
batch_size,
column_names=column_names,
label_name=label_name,
um_epochs=1)
def pack_features_vector(features, labels):
    """Pack the features into a single array."""
    features = tf.stack(list(features.values()), axis=1)
    return features, labels
train_dataset = train_dataset.map(pack_features_vector)
model = tf.keras.Sequential([tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,), # input shape required
tf.keras.layers.Dense(10, activation=tf.nn.relu),
tf.keras.layers.Dense(3)])
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
def loss(model, x, y):
y_ = model(x)
return loss_object(y_true=y, y_pred=y_)
def grad(model, inputs, targets):
with tf.GradientTape() as tape:
    loss_value = loss(model, inputs, targets)
    return loss_value, tape.gradient(loss_value, model.trainable_variables)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
## Note: Rerunning this cell uses the same model variables
# Keep results for plotting
train_loss_results = []
train_accuracy_results = []
num_epochs = 201
for epoch in range(num_epochs):
    epoch_loss_avg = tf.keras.metrics.Mean()
    epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()
    for x, y in train_dataset:
        # Optimize the model
        loss_value, grads = grad(model, x, y)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
        # Track progress
        epoch_loss_avg(loss_value)  # Add current batch loss
        # Compare predicted label to actual label
        epoch_accuracy(y, model(x))
    # End epoch
    train_loss_results.append(epoch_loss_avg.result())
    train_accuracy_results.append(epoch_accuracy.result())
    if epoch % 50 == 0:
        print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch, epoch_loss_avg.result(), epoch_accuracy.result()))
Listing 25: Final code for all pieces of the model