Advanced Learning Algorithms

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023

[1] The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Neural Network Inference

The original motivation behind neural networks was to replicate how the human brain learns and thinks; now it has significantly diverged from that. In the human brain, neurons receive, process, and transmit electrical impulses. Often, an individual neuron aggregates inputs from multiple other neurons before generating an output. The artificial neural network utilises a mathematical model of what a neuron does: a neuron takes an input, process it, and sends it to another neuron. When building a model, rather than building a single neuron at a time, we simulate many such neurons simultaneously.

Neural Networks have exploded in recent years because, as opposed to traditional AI, neural networks can keep on improving on accuracy the more training samples they receive.
Let us illustrate the way neural networks work through an example, *demand prediction*: Demand prediction will take several features of a product as an input (placed in the *input layer*), and generate an output that says whether or not the product will be a top–seller (placed in the *output layer*. A neuron is an element that will take certain inputs, compute a value, and send that value to the next layer of neurons for further processing. Imagine you are given a piece of clothing as an input. In the first layer, as an input, you will have price, cost of shipping, marketing, and material quality. You might suspect that the ways a piece of clothing becomes a top seller are several: the affordability (combination of shipping cost and pricing), awareness (function of marketing), and perception of quality (function of material quality and pricing). We are going to feed the required inputs from the input layer to a layer with three neurons that will calculate each of these. For instance, the first neuron in this second layer could use logistic regression with prices/shipping costs where the output is, do people think this is affordable? This layer (of which a neural network can have an infinite amount) is called the *hidden layer*. The intermediate values calculated by the hidden layers (e.g. affordability and awareness) are also called **activations**. Lastly, the output layer neuron will utilise the information generated by the neurons in the hidden layer and generate an output regarding the popularity of a given product. It is important to notice that all the neurons in each layer will have access to all the output information from the previous

layer. Because of this, the input and output can be represented as vectors of values $(\vec{X}, \vec{Y})$.

A good way of thinking of a neural network is that it's an automated version of feature engineering: the relationship between the last hidden layer and the output layer in this example is just logistic regression but it has more efficient functions that lead to better predictions. When training a neural network, it is important to notice that it will figure out the best features to use all by itself — no need for you to guide the model in any direction!
A similar idea can be applied to computer vision (CV). An image is made up of pixels, which can be seen as matrices of pixel elements, each containing information about its brightness, colour... What if we unroll this matrix into a feature vector and feed it into a neural network? The first hidden layer will extract some features from the input, which will be fed as an output vector to the next layer, and so on, until you reach an output layer that gives you the probability of it being a person x. Each layer will be focusing in something different: one might be focusing in detection of the bottom of an ear, curves in the face, eyes, etc.

Neural network layers are the fundamental blocks of modern neural networks. Quantities related to each layer are denoted with a bracketed superscript. For example, the output and activating values of the first layer of a network are $\vec{a}^{[1]}$ and $\vec{w}^{[1]}, b^{[1]}, a^{[1]}$ respectively. When counting the layers of a neural network, we usually include the output layer but not the input layer in the count. Generalising, the activation value of layer l, neuron j is $a_j^{[l]} = g(\vec{w}_j \cdot \vec{a}^{l-1} + b_j^l)$, where $g()$ is the *activation function*. This could be the sigmoid function $g(x) = \frac{1}{1+e^{-x}}$ or other functions that shall be introduced later.
Forward propagation is a common algorithm to make predictions. It consists of going from the input layer to the output layer on a layer–by–layer basis. This is in contrast to backward propagation, which is often used for training and we shall cover later on. Here is a simple neural network implemented with Python, Tensorflow, and Keras:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras import Sequential
from tensorflow.keras.losses import MeanSquaredError,
    BinaryCrossentropy
from tensorflow.keras.activations import sigmoid
from lab_utils_common import dlc
from lab_neurons_utils import plt_prob_1d, sigmoidnp, plt_linear,
    plt_logistic
plt.style.use('./deeplearning.mplstyle')
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)

X_train = np.array([[1.0], [2.0]], dtype=np.float32)           #(
    size in 1000 square feet)
Y_train = np.array([[300.0], [500.0]], dtype=np.float32)       #(
    price in 1000s of dollars)

#Let us define a layer with one neuron and compare it to linear
```

```
    regression
linear_layer = tf.keras.layers.Dense(units=1, activation = 'linear',
    )

#the input layer must be in 2-D, so we need to reshape it before
    passing it into linear layer
a1 = linear_layer(X_train[0].reshape(1,1))
#When we instantiate it like this, the weight and bias will be
    initialised to random values
w, b= linear_layer.get_weights()
#Let us set them to definite values
set_w = np.array([[200]])
set_b = np.array([100])

# set_weights takes a list of numpy arrays
linear_layer.set_weights([set_w, set_b])

#Notice that now, these two will have the same output!
a1 = linear_layer(X_train[0].reshape(1,1))
print(a1)
alin = np.dot(set_w,X_train[0].reshape(1,1)) + set_b
print(alin)

prediction_tf = linear_layer(X_train)
prediction_np = np.dot( X_train, set_w) + set_b
```

1.1 TensorFlow implementation

Imagine we are trying to determine whether or not the quality of roasted coffee is good based on the time it was cooked and the temperature it was used. We will do this using TensorFlow the following way:

```
'''
Not covered here:
-How to load libraries
-How to load parameters w and b
'''
x = np.array([[200.0, 17.0]])
#Dense refers to the type of layer of the neural network we have
    learnt so far. There are other types.
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
layer_2 = Dense(units = 1, activation = 'sigmoid')
a2 = layer_2(a1)

#thresholding
if a2>= 0.5:
    yhat = 1
else:
    yhat = 0
```

There are some inconsistencies in how TensorFlow and NumPy handle data. When you write `x = np.array([200, 17])`, you are creating a $1x2$ matrix; however, if you write

`x = np.array([200, 17])`, it results in a 1D vector instead of a 2D matrix. With TensorFlow, the convention is to represent data as tensors. This is because that way, it can compute large data more efficiently. You can convert TensorFlow tensors into NumPy array with the function `tensorname.numpy()`.
Forward propagation and learning can be implemented differently through TensorFlow: instead of passing the data manually from each layer, you can string different layers into one network through the sequential framework:

```
#Define layers as before
layer_1 = Dense(units=3, activation='sigmoid')
layer_2 = Dense(units = 1, activation = 'sigmoid')
#use the sequential framework
model = Sequential([layer_1, layer_2])
```

Two finally be able to utilise this network you need to use two methods: `model.compile(...)` which we will cover later, and `model.fit(x,y)`, where x and y are NumPy tensors for the input and output respectively. Lastly, the method `model.predict(x_new)` will do the forward propagation for you.
Usually, the previous implementation is shortened to:

```
model = Sequential([Dense(units=3, activation="sigmoid"),Dense(units
    =1, activation="sigmoid")])
```

Although this is hardly standard practise when it comes to real–life implementation of neural networks, let us learn how to code a neural network from scratch using Python so that we can deepen our knowledge about what is happening in each layer.

```
import numpy as np
import tensorflow as tf
from lab_utils_common import dlc, sigmoid
from lab_coffee_utils import load_coffee_data, plt_roast, plt_prob,
    plt_layer, plt_network, plt_output_unit
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)

X,Y = load_coffee_data();
#pre-existing dataset
#For reference: print(X.shape, Y.shape) --> (200, 2) (200, 1)

#Normalise the data
norm_l = tf.keras.layers.Normalization(axis=-1)
norm_l.adapt(X)  # learns mean, variance
Xn = norm_l(X)

# Define the activation function
g = sigmoid

#Define the function to compute the activations of a dense layer
def my_dense(a_in, W, b):
    """
    Computes dense layer
    Args:
```

```
      a_in (ndarray (n, )) : Data, 1 example
      W    (ndarray (n,j)) : Weight matrix, n features per unit, j units
      b    (ndarray (j, )) : bias vector, j units
    Returns
      a_out (ndarray (j,))  : j units|
    """
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = W[:,j]
        z = np.dot(w, a_in) + b[j]
        a_out[j] = g(z)
    return(a_out)

#the following function builds a two-layer neural network using the function above
def my_sequential(x, W1, b1, W2, b2):
    a1 = my_dense(x,  W1, b1)
    a2 = my_dense(a1, W2, b2)
    return(a2)
#Let us create a function to predict
def my_predict(X, W1, b1, W2, b2):
    m = X.shape[0]
    p = np.zeros((m,1))
    for i in range(m):
        p[i,0] = my_sequential(X[i], W1, b1, W2, b2)
    return(p)
```

Vectorisation has allowed neural networks to be scaled up and improved significantly, especially because this allows parallel computing hardware to be utilised the most efficiently. For instance, we can improve on the previously introduced activation-calculator as follows:

```
'''
Import relevant libraries
'''
X = np.array([200, 17]) #2d array
W = np.array([[1, -3, 5], [-2, 4, -6]]) #2d array
B = np.array([[-1, 1, 2]]) #1x3 2D array

def dense(A_in, W, B):
    Z = np.matmul(A_in,W) + B #vectorised to avoid for-loop
    A_out = g(Z) #apply the activation function
    return A_out
```

2 Neural Network Training

The code you normally would use in TensorFlow to train a network is the following:

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

```

```
#Sequentially string together layers of neural network
model = Sequential([Dense(units=25, activation='sigmoid'), Dense(
    units=15, activation='sigmoid'), Dense(units = 1, activation = '
    sigmoid')])

from tensorflow.keras.losses import BinaryCrossentropy

#Ask TensorFlow to compile the model. Key: specify the loss function
model.compile(loss=BinaryCrossentropy())

#Call the fit function. It fits the dataset X,Y into the model
    specified earlier using the loss
#function added in the compile step
model.fit(X,Y,epochs=100)
#Epoch refers to the number of steps in gradient descent
```

Let us understand in detail what is happening in each of these steps.
Let us recall the steps of logistic regression in the previous course:

1. Specify how to compute the output given input x and parameters w, b
2. Specify the loss and cost functions to be utilised
3. Use gradient descent to minimise the cost as a function of parameters $\vec{w}$ and b

These same three steps are valid for training neural networks:

1. Specify how different layers of the neural network should be stringed together.
2. Compile the model and tell it which loss/cost to use. The most commonly used one is *logistic loss*, also known as *binary cross entropy*. This is the same as the one used for logistic regression $(L(f(\vec{x}, y)) = -y\,log(f(\vec{x})) - (1-y)log(1-f(\vec{x})))$. There are a lot more loss functions available; for instance, if you are working on a regression problem, you can call `model.compile(loss = MeanSquaredError())`.
3. Call the `.fit` method to minimise the cost function that uses the loss specified in the previous step. This will trigger gradient descent; namely, repeat updates to parameters until convergence. The key part of this algorithm is computing the partial derivative of each weight, and TensorFlow (as well as most other libraries) do this through *backpropagation*. There are actually some algorithms that are more efficient than gradient descent, which we shall see later.

2.1 Activation functions

Up to this point, the only activation function we have covered is the sigmoid function. This is however not the best function we can use. Rather than modelling functions as either outputting one or zero, we can model them as, for instance, outputting a non–negative number. This function is what is called the *ReLU*, which stands for rectified linear unit. There is also the *linear activation function*, which is only $g(z) = z$. When this one is used, sometimes it is said that no activation function is used.

Depending on what the target label is, there is a fairly natural function for the output layer. The process of choosing this function can get a little trickier when it comes to the activation function of the hidden layers.
For the output layer:

- Binary Classification: Use the sigmoid function
- Regression (positive and negative values): Use the linear activation function
- Regression with only positive values: ReLU

For the hidden layer, the most common function by far is ReLU. This is because ReLU is faster to compute. Additionally, it only *goes flat* in one part of the graph — the negative side. The sigmoid function, on the other hand, flattens at two sides. This flattening slows down the learning.
There are a lot more activation functions, such as the *tan h activation function, leaky ReLU, swish*, among others; however, they are beyond the scope of our discussion in this document.
We need activation functions because without it, the networks would only be able to compute linear regression — not anything more complex. Remember that a linear function of a linear function is a linear function.

2.2 Multiclass Classification

Multiclass classification refers to classification where you can have more than just two possible outputs.
The Softmax Regression Algorithm is a generalisation of the logistic regression algorithm that allows us to carry out multiclass classification. The output of logistic regression is how likely an input $\vec{x}$ is to be a member of a class a_1. However, we can also say the output gives us the probability it is in a_2, since the sum of both probabilities must be 1. Generalising this, if we have n possible outputs, the function will compute $z_1 = \vec{w}_1 \cdot \vec{x} + b_1$, $z_2 = \vec{w}_2 \cdot \vec{x} + b_2$, $\ldots$, $z_n = \vec{w}_n \cdot \vec{x} + b_n$. Then, it will pass z into the function $a_i = \frac{e^{z_i}}{e^{z_1}+e^{z_2}+\ldots+e^{z_n}}$ This is called softmax regression.
The cost function for softmax regression will also change: instead of $-y\,log\,a_1 - (1-y)log(1-a_1)$, we will use:

$$loss(a_1, a_2, \cdots, a_N, y) = \begin{cases} -log(a_1), & y = 1, \\ -log(a_2), & y = 2, \\ \ldots & \\ -log(a_N), & y = N. \end{cases}$$

Usually, the activation for the output layer is the one that is computed using the softmax regression function. We will compute z regularly as $\vec{w} \cdot \vec{a} + b$ for every layer, and then to get each a we need to plug these z values into the softmax activation function. On TensorFlow, this would be implemented as

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

#specify the model
model = Sequential([Dense(units=25, activation = 'relu'), Dense(
    units=15, activation='relu'), Dense(units=10, activation='softmax
    ')])

#specify loss and cost
from tensorflow.keras.losses import SparseCategoricalCrossentropy

#from_logits=True gives TensorFlow freedom in deciding how to
    compute --> higher numerical accuracy
model.compile(loss=SparseCategoricalCrossentropy(from_logits=True))

#train on data to minimise the cost function
model.fit(X,Y, epochs=100)

#Fit
logits = model(X)

#predict
f_x = tf.nn.sigmoid(logit)
```

Multi–label Classification refers to looking for several labels in a single input — commonly used in Computer Vision. This is not the same as multi–class clasification.

Up to now, we have been relying on gradient descent to minimise the cost function; there are however more efficient algorithms available. One of the most commonly used is **Adam** (Adaptive Moment Estimation). As you might have noticed, gradient descent will always go in the same direction on every step. Adam will take advantage of this fact and automatically increase α: it will see if the algorithm keeps taking very small steps in the same direction and increase the value of alpha if that is the case. Adam also does the opposite: if the algorithm keeps oscillating back and forth, it means that its learning rate is too large, so Adam will adjust it.
Interestingly, Adam does not use a global value for α but rather different rates for every single parameter in the model. In code, Adam is implemented as:

```
#model
model = Sequential([tf.keras.layers.Dense(units = 25, activation='
    sigmoid'), tf.keras.layers.Dense(units=15, activation = 'sigmoid'
    ), tf.keras.layers.Dense(units=10, activation='linear')])
#compile
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3)
    , loss = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True))
#fit
model.fit(X,Y,epochs=100)
```

So far, we have only utilised the `dense` neuron layer type. The main feature of this type of layer is every single one of its neurons gets all the activations from the previous layer as

an input. There are however several other types:

- Convolutional Layer: each neuron only looks at a part of the previous layer's inputs. This leads to faster computation and to need less training data — since the network will be less prone to overfitting.
- Fully Connected
- Pooling
- Normalisation
- Recurrent

Here is an example of a neural network for hand written digit recognition:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import linear, relu, sigmoid


import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)

from public_tests import *

from autils import *
from lab_utils_softmax import plt_softmax
np.set_printoptions(precision=2)

plt_act_trio()

def my_softmax(z):
    """ Softmax converts a vector of values to a probability
    distribution.
    Args:
      z (ndarray (N,))  : input data, N features
    Returns:
      a (ndarray (N,))  : softmax of z
    """

    ez = np.exp(z)
    a = ez/np.sum(ez)

    return a

z = np.array([1., 2., 3., 4.])
a = my_softmax(z)
atf = tf.nn.softmax(z)

#Borrowing from a dataset available online
```

```
X, y = load_data()

#represent the model
tf.random.set_seed(1234) # for consistent results
model = Sequential(
    [
        tf.keras.layers.InputLayer((400,)),
        tf.keras.layers.Dense(25, activation="relu", name="L1"),
        tf.keras.layers.Dense(15, activation="relu", name="L2"),
        tf.keras.layers.Dense(10, activation="linear", name="L3")
    ], name = "my_model"
)

[layer1, layer2, layer3] = model.layers

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
    True),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
)

history = model.fit(
    X,y,
    epochs=40
)

mage_of_two = X[1015]
display_digit(image_of_two)

prediction = model.predict(image_of_two.reshape(1,400))  #
    prediction

print(f" predicting a Two: \n{prediction}")
print(f" Largest Prediction index: {np.argmax(prediction)}")

yhat = np.argmax(prediction_p)

print(f"np.argmax(prediction_p): {yhat}")
```

3 Practical Advice for Building Machine Learning Systems

Having a systematic way to evaluate your model can lead you to target and improve the areas that might be hindering it. One technique is to split your dataset into groups. One can be a *training set* and the other one can be a *test set*. The purpose of this is to assess the model's accuracy from being trained with the training set by using the test set. This is done by computing the average error of the test set. This is the same of computing the test error. If the test cost is high, this means that the model is not good at generalising.
We have seen that once the parameters $\vec{w}$ and b have been fit to the training set, the training error may not be a good indicator of how well the algorithm will generalise to new examples that are outside the training set. If you want to automatically select a model for your

machine learning model is by splitting your data into three sets instead of two: the training set, the testing set, and the *cross–validation set*. Cross–validation refers to it being an extra set we use to check the validity of different models. Once we have this, we need to compute the training, cross–validation, and test errors, not including the regularisating term that is included during training.

Armed with these three values, we can do model selection by fitting the possible parameters from the models we are considering into the cross–validation set and compute the cost function. The one that outputs the lowest cost is the most efficient. This rationale also works for neural networks — train different networks and compare their performance with the cost function of the cross–validation set.

It is standard practise to make decisions on the model only using the training and cross–validation sets; only once you have come up with a final model will you evaluate it to the test set. This guarantees that the test set is a fair estimate of how good the model will generalise to new data.

Take a look at this example on splitting datasets into the three aforementioned subsets, evaluating regression and classification models, adding polynomial features to improve the performance of a linear regression model, and comparing several neural network architectures:

```
# for array computations and loading data
import numpy as np

# for building linear regression models and preparing data
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# for building and training neural networks
import tensorflow as tf

# custom functions
import utils

# reduce display precision on numpy arrays
np.set_printoptions(precision=2)

# suppress warnings
tf.get_logger().setLevel('ERROR')
tf.autograph.set_verbosity(0)

#developing a model for a regression problem
#Given a dataset of 50 examples with an input feature x and
    corresponding target y
# Load the dataset from the text file
data = np.loadtxt('./data/data_w3_ex1.csv', delimiter=',')

# Split the inputs and outputs into separate arrays
x = data[:,0]
y = data[:,1]

```

```
# Convert 1-D arrays into 2-D because the commands later will require it
x = np.expand_dims(x, axis=1)
y = np.expand_dims(y, axis=1)

#Split the dataset
# Get 60% of the dataset as the training set. Put the remaining 40% in temporary variables: x_ and y_.
x_train, x_, y_train, y_ = train_test_split(x, y, test_size=0.40, random_state=1)

# Split the 40% subset above into two: one half for cross validation and the other for the test set
x_cv, x_test, y_cv, y_test = train_test_split(x_, y_, test_size=0.50, random_state=1)

# Delete temporary variables
del x_, y_

#Scale all the features for smooth processing
# Initialize the class
scaler_linear = StandardScaler()

# Compute the mean and standard deviation of the training set then transform it
X_train_scaled = scaler_linear.fit_transform(x_train)

print(f"Computed mean of the training set: {scaler_linear.mean_.squeeze():.2f}")
print(f"Computed standard deviation of the training set: {scaler_linear.scale_.squeeze():.2f}")

#train the model
# Initialize the class
linear_model = LinearRegression()

# Train the model
linear_model.fit(X_train_scaled, y_train )

#Evaluate the model's performance using the MSE for training and crossvalidation sets
# Feed the scaled training set and get the predictions
yhat = linear_model.predict(X_train_scaled)

# Use scikit-learn's utility function and divide by 2
print(f"training MSE (using sklearn function): {mean_squared_error(y_train, yhat) / 2}")

# for-loop implementation
total_squared_error = 0

for i in range(len(yhat)):
    squared_error_i  = (yhat[i] - y_train[i])**2
    total_squared_error += squared_error_i
```

```
76
77 mse = total_squared_error / (2*len(yhat))
78
79 print(f"training MSE (for-loop implementation): {mse.squeeze()}")
80
81 # Scale the cross validation set using the mean and standard
       deviation of the training set
82 X_cv_scaled = scaler_linear.transform(x_cv)
83
84 print(f"Mean used to scale the CV set: {scaler_linear.mean_.squeeze
       ():.2f}")
85 print(f"Standard deviation used to scale the CV set: {scaler_linear.
       scale_.squeeze():.2f}")
86
87 # Feed the scaled cross validation set
88 yhat = linear_model.predict(X_cv_scaled)
89
90 # Use scikit-learn's utility function and divide by 2
91 print(f"Cross validation MSE: {mean_squared_error(y_cv, yhat) / 2}")
92
93 #Add polynomial features
94 # Instantiate the class to make polynomial features
95 poly = PolynomialFeatures(degree=2, include_bias=False)
96
97 # Compute the number of features and transform the training set
98 X_train_mapped = poly.fit_transform(x_train)
99
100 # Instantiate the class
101 scaler_poly = StandardScaler()
102
103 # Compute the mean and standard deviation of the training set then
       transform it
104 X_train_mapped_scaled = scaler_poly.fit_transform(X_train_mapped)
105
106 # Preview the first 5 elements of the scaled training set.
107 print(X_train_mapped_scaled[:5])
108
109 # Initialize the class
110 model = LinearRegression()
111
112 # Train the model
113 model.fit(X_train_mapped_scaled, y_train )
114
115 # Compute the training MSE
116 yhat = model.predict(X_train_mapped_scaled)
117 print(f"Training MSE: {mean_squared_error(y_train, yhat) / 2}")
118
119 # Add the polynomial features to the cross validation set
120 X_cv_mapped = poly.transform(x_cv)
121
122 # Scale the cross validation set using the mean and standard
       deviation of the training set
123 X_cv_mapped_scaled = scaler_poly.transform(X_cv_mapped)
124
```

```
# Compute the cross validation MSE
yhat = model.predict(X_cv_mapped_scaled)
print(f"Cross validation MSE: {mean_squared_error(y_cv, yhat) / 2}")

# Initialize lists containing the lists, models, and scalers
train_mses = []
cv_mses = []
models = []
scalers = []

# Loop over 10 times. Each adding one more degree of polynomial
    higher than the last.
for degree in range(1,11):

    # Add polynomial features to the training set
    poly = PolynomialFeatures(degree, include_bias=False)
    X_train_mapped = poly.fit_transform(x_train)

    # Scale the training set
    scaler_poly = StandardScaler()
    X_train_mapped_scaled = scaler_poly.fit_transform(X_train_mapped
    )
    scalers.append(scaler_poly)

    # Create and train the model
    model = LinearRegression()
    model.fit(X_train_mapped_scaled, y_train )
    models.append(model)

    # Compute the training MSE
    yhat = model.predict(X_train_mapped_scaled)
    train_mse = mean_squared_error(y_train, yhat) / 2
    train_mses.append(train_mse)

    # Add polynomial features and scale the cross validation set
    poly = PolynomialFeatures(degree, include_bias=False)
    X_cv_mapped = poly.fit_transform(x_cv)
    X_cv_mapped_scaled = scaler_poly.transform(X_cv_mapped)

    # Compute the cross validation MSE
    yhat = model.predict(X_cv_mapped_scaled)
    cv_mse = mean_squared_error(y_cv, yhat) / 2
    cv_mses.append(cv_mse)

# Get the model with the lowest CV MSE (add 1 because list indices
    start at 0)
# This also corresponds to the degree of the polynomial added
degree = np.argmin(cv_mses) + 1
print(f"Lowest CV MSE is found in the model with degree={degree}")

# Add polynomial features to the test set
poly = PolynomialFeatures(degree, include_bias=False)
X_test_mapped = poly.fit_transform(x_test)

```

```
# Scale the test set
X_test_mapped_scaled = scalers[degree-1].transform(X_test_mapped)

# Compute the test MSE
yhat = models[degree-1].predict(X_test_mapped_scaled)
test_mse = mean_squared_error(y_test, yhat) / 2

print(f"Training MSE: {train_mses[degree-1]:.2f}")
print(f"Cross Validation MSE: {cv_mses[degree-1]:.2f}")
print(f"Test MSE: {test_mse:.2f}")


'''
Let
Us
Work
on
Neural
Networks
'''

#Prepare the data
# Add polynomial features
degree = 1
poly = PolynomialFeatures(degree, include_bias=False)
X_train_mapped = poly.fit_transform(x_train)
X_cv_mapped = poly.transform(x_cv)
X_test_mapped = poly.transform(x_test)

#Scale Input
# Scale the features using the z-score
scaler = StandardScaler()
X_train_mapped_scaled = scaler.fit_transform(X_train_mapped)
X_cv_mapped_scaled = scaler.transform(X_cv_mapped)
X_test_mapped_scaled = scaler.transform(X_test_mapped)

# Initialize lists that will contain the errors for each model
nn_train_mses = []
nn_cv_mses = []

# Build the models
nn_models = utils.build_models()

# Loop over the the models
for model in nn_models:

    # Setup the loss and optimizer
    model.compile(
    loss='mse',
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.1),
    )

    print(f"Training {model.name}...")

```

```
    # Train the model
    model.fit(
        X_train_mapped_scaled, y_train,
        epochs=300,
        verbose=0
    )

    print("Done!\n")


    # Record the training MSEs
    yhat = model.predict(X_train_mapped_scaled)
    train_mse = mean_squared_error(y_train, yhat) / 2
    nn_train_mses.append(train_mse)

    # Record the cross validation MSEs
    yhat = model.predict(X_cv_mapped_scaled)
    cv_mse = mean_squared_error(y_cv, yhat) / 2
    nn_cv_mses.append(cv_mse)


# print results
print("RESULTS:")
for model_num in range(len(nn_train_mses)):
    print(
        f"Model {model_num+1}: Training MSE: {nn_train_mses[
    model_num]:.2f}, " +
        f"CV MSE: {nn_cv_mses[model_num]:.2f}"
        )
 # Select the model with the lowest CV MSE
model_num = 3

# Compute the test MSE
yhat = nn_models[model_num-1].predict(X_test_mapped_scaled)
test_mse = mean_squared_error(y_test, yhat) / 2

print(f"Selected Model: {model_num}")
print(f"Training MSE: {nn_train_mses[model_num-1]:.2f}")
print(f"Cross Validation MSE: {nn_cv_mses[model_num-1]:.2f}")
print(f"Test MSE: {test_mse:.2f}")


'''
Let
Us
Work
on
Classification
'''
#load preexisting dataset
# Load the dataset from a text file
data = np.loadtxt('./data/data_w3_ex2.csv', delimiter=',')

# Split the inputs and outputs into separate arrays
```

```
x_bc = data[:,:-1]
y_bc = data[:,-1]

# Convert y into 2-D because the commands later will require it (x is already 2-D)
y_bc = np.expand_dims(y_bc, axis=1)

#Split the dataset
from sklearn.model_selection import train_test_split

# Get 60% of the dataset as the training set. Put the remaining 40% in temporary variables.
x_bc_train, x_, y_bc_train, y_ = train_test_split(x_bc, y_bc, test_size=0.40, random_state=1)

# Split the 40% subset above into two: one half for cross validation and the other for the test set
x_bc_cv, x_bc_test, y_bc_cv, y_bc_test = train_test_split(x_, y_, test_size=0.50, random_state=1)

# Delete temporary variables
del x_, y_

# Scale the features

# Initialize the class
scaler_linear = StandardScaler()

# Compute the mean and standard deviation of the training set then transform it
x_bc_train_scaled = scaler_linear.fit_transform(x_bc_train)
x_bc_cv_scaled = scaler_linear.transform(x_bc_cv)
x_bc_test_scaled = scaler_linear.transform(x_bc_test)

#Evaluate the error
# Sample model output
probabilities = np.array([0.2, 0.6, 0.7, 0.3, 0.8])

# Apply a threshold to the model output. If greater than 0.5, set to 1. Else 0.
predictions = np.where(probabilities >= 0.5, 1, 0)

# Ground truth labels
ground_truth = np.array([1, 1, 1, 1, 1])

# Initialize counter for misclassified data
misclassified = 0

# Get number of predictions
num_predictions = len(predictions)

# Loop over each prediction
for i in range(num_predictions):

```

```
    # Check if it matches the ground truth
    if predictions[i] != ground_truth[i]:

        # Add one to the counter if the prediction is wrong
        misclassified += 1

# Compute the fraction of the data that the model misclassified
fraction_error = misclassified/num_predictions

#building and training the model
# Initialize lists that will contain the errors for each model
nn_train_error = []
nn_cv_error = []

# Build the models
models_bc = utils.build_models()

# Loop over each model
for model in models_bc:

    # Setup the loss and optimizer
    model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
    )

    print(f"Training {model.name}...")

    # Train the model
    model.fit(
        x_bc_train_scaled, y_bc_train,
        epochs=200,
        verbose=0
    )

    print("Done!\n")

    # Set the threshold for classification
    threshold = 0.5

    # Record the fraction of misclassified examples for the training
     set
    yhat = model.predict(x_bc_train_scaled)
    yhat = tf.math.sigmoid(yhat)
    yhat = np.where(yhat >= threshold, 1, 0)
    train_error = np.mean(yhat != y_bc_train)
    nn_train_error.append(train_error)

    # Record the fraction of misclassified examples for the cross
    validation set
    yhat = model.predict(x_bc_cv_scaled)
    yhat = tf.math.sigmoid(yhat)
    yhat = np.where(yhat >= threshold, 1, 0)
    cv_error = np.mean(yhat != y_bc_cv)
```

```
    nn_cv_error.append(cv_error)

# Print the result
for model_num in range(len(nn_train_error)):
    print(
        f"Model {model_num+1}: Training Set Classification Error: {
    nn_train_error[model_num]:.5f}, " +
        f"CV Set Classification Error: {nn_cv_error[model_num]:.5f}"
        )

 #pick
 # Select the model with the lowest error
model_num = 3

# Compute the test error
yhat = models_bc[model_num-1].predict(x_bc_test_scaled)
yhat = tf.math.sigmoid(yhat)
yhat = np.where(yhat >= threshold, 1, 0)
nn_test_error = np.mean(yhat != y_bc_test)

print(f"Selected Model: {model_num}")
print(f"Training Set Classification Error: {nn_train_error[model_num
    -1]:.4f}")
print(f"CV Set Classification Error: {nn_cv_error[model_num-1]:.4f}"
    )
print(f"Test Set Classification Error: {nn_test_error:.4f}")
```

Looking at the bias and variance of a model may allow us to understand why it is not performing as we want to. Identifying high bias is straightforward: the cost of the training set will be high, as well as the cross–validation cost. If the cost of the training set is low but the cost of the cross–validation set is high, it may be a case of high variance.
In general, when you fit a higher order polynomial, the error of the training set will decrease. On the other hand, the cost of the cross–validation set will be a concave function: the minimum point will be a polynomial number that is not the highest but also not the lowest. Our objective is to find this value.
Cross–validation can also help us choose a good value of the regularisation parameter λ. We will calculate the cost of the cross–validation set using several lambda values within a range. The value that results in the lowest value for lambda is the ideal one for our model.

When judging if the training error is high, it's useful to establish a baseline level of performance. A common way of doing this is to measure how well humans normally perform, how competing algorithms perform, and guess based on experience. Look at the gaps between the baseline performance, the training error, and the cross–validation error. If the former is big, it is a bias problem. If the latter is big, it is a variance problem. Looking at these numbers, we can intuitively get a sense of the type of problem we have.

Learning curves are a way to understand how the learning algorithm is doing as a function of the amount of experience it has (e.g. the number of training examples it has). An interesting relation is that, the larger the training set size, the smaller the cross–validation

error and the larger the training error. Intuitively, the larger the training set, the more difficult it is for a function to fit all the samples perfectly. Another important fact is that the cross–validation error tends to be higher than the training error.

Troubleshooting:

- Get more training examples: helps fix high variance.
- Try smaller sets of features: too many features gives too much flexibility for your model to become very specific to the data. This helps to fix high variance.
- Try getting additional features: helps fix high bias.
- Try adding polynomial features: helps fix high bias.
- Try decreasing λ: means pay less attention to the regularisation term and more attention to the cost function, so it fixes bias.
- Try increasing λ: the converse of the above — forces the algorithm to fit a smoother, less wiggly line; fixes high variance.

One of the reasons why neural networks have been so successful is how they present new ways for us to deal with variance and bias. Normally, we have a bias–variance tradeoff: you have to balance the complexity that is the degree of the polynomial and the regularisation parameter. Neural networks allow us to exit this dilemma; if bias is high, just use a bigger neural network (more hidden layers or more hidden units per layer). If the variance is high, get more data and retrain the model. There are however limitations to this method, since a larger neural network is computationally expensive. Furthermore, it is sometimes not easy to get more data.
With proper regularisation, the risk of overfitting as the neural network grows larger does not increase.

3.1 Machine Learning Development Process

Iterative Loop of Machine Learning Development:
Usually you start by deciding what is the overall architecture of your system. Given those decisions, you implement and train the model. The next step is to look at diagnostics (e.g. bias, variance), and based on the results of the diagnostics, go back to step one and revise the overall architecture. It often takes several iterations of this loop for an acceptable model to be produced.
Another diagnostics you can look at is *error analysis*: manually examine examples that the algorithm has misclassified, group them based on common traits, and see which big issues deserve your focus.
Adding Data:
When training machine learning algorithms, sometimes it's tempting to just add more data, but this is a process that is costly in both time and other resources. A better approach is to

add more data of the types where error analysis have indicated it might help.
A way to obtain new data when it is hard to get completely new samples is through a technique named *data augmentation*, which involves modifying an existing training example (e.g. distorting, transforming, zooming, cropping) to create a new example that helps the algorithm improve its capabilities to identify a particular label. It is not helpful to just add random noise; you have to make sure that after adding the noise you still end up with something similar enough to what you want.
A similar technique is *data synthesis*, which is creating brand new examples as opposed to modifying existing ones. Another powerful technique is *transfer learning*. The key idea behind it is to take data from a totally different, barely related task and process it through a neural network in a way that improves your model's performance. This does not apply to everything, but can be powerful when it does.
Copy a pre–existing, trained neural network and only change the output layer. Transfer learning means using the parameters of the already trained network as starting points for the training of your neural network. Usually, transfer learning involves two possibilities, only training the output layer parameters (ideal when not much data is available) or training all parameters but use the previous weights as a starting point (ideal when a large dataset is available). Often, this is also called *supervised pre–training*.
An important restriction for transfer learning is that the pre–trained model must use the same input type. There are two key steps in the process:

1. Download neural network parameters pretrained on a large dataset with the same input type as your application
2. Further train (fine tune) the network on your own data

The full cycle of a machine learning project:

1. Scope project: define what you want to do.
2. Collect data: define and collect the data.
3. Train model: training, error analysis, iterative improvement. The improvement may involve collecting more data.
4. Deploy in production environment: also involves monitoring and maintain the system. A common way to deploy a model is to host it on an inference server. Any app or service that will utilise the machine learning model will perform an API call to the server and receive the inference.

3.2 Skewed Datasets

A skewed dataset is one where significantly more elements belong to one category than the other, usual error metrics like accuracy are not adequate. A common pair of error metrics are precision and recall. Precision involves counting how many examples in each class were wrongly and correctly classified. In binary classification, the terms true positive, false positive, true negative, and false negative are commonly used; The true or false refer to whether

or not the output label matches to the actual label the sample is. Precision is measured in terms of a particular category as $\frac{\text{true positives}}{\text{true positives+false positives}}$. Recall refers to what percentage of all the elements of a category in your dataset did your algorithm correctly classify into the right category. This is expressed as $\frac{\text{True positives}}{\text{True positives + False negatives}}$.
High precision that if the algorithm outputs a positive, it is likely to be a correct result; high recall that if the element is a member of the class, the algorithm is likely to correctly classify it. Precision is increased when the threshhold is increased (e.g. requiring 0.9 confidence before outputting a positive). This, however, reduces the recall. The opposite is also true: lowering the threshhold increases the recall but reduces the precision. If you want to automatically compare precision and recall is using the F1 score, a combination of precision and recall that pays more attention to whichever value is lower. This is also known in mathematics as the *harmonic mean*.

4 Decision Trees

Every type we receive a new example, the algorithm will start at the root, and then based on a certain computation, it will go down to the child node up until it gets to a leaf, which allows the model to perform an inference. The inner nodes of a decision tree are called decision nodes. Several decision trees can be created for a specific inference process; we are going to learn of an algorithm that will decide this for us.
Given a set of training examples, the first thing we need to do is to choose a feature to place at the root node. Then, we will analyse the examples that get classified into each decision node, and create another layer that allows us to further subdivide the set. Some key decisions are how to choose what feature to split on at each node — our goal is to maximise purity, which results on getting a set that ends in the most positive classifications. The algorithm will iteratively choose which feature to use by comparing different possibilities and checking their purity. Another important decision is, when do we stop splitting? This is usually decided when either a node is 100% one class, when purity improvements are below a threshold, when the number of examples in a node is below a threshold, and when splitting the node results in the tree exceeding a maximum depth.
Entropy (H) is commonly used to measure impurity; the smaller the entropy function is, the better. The way this is calculated when there are only two categories is $H(p_1) = -p_1 log_2(p_1) - p_0 log_2(p_0)$, where p_1 is the fraction of examples belonging to one category and $p_0 = 1 - p_1$. Other functions, such as *the Gini criteria* also work, but we will focus on entropy in these notes.
The way we choose what feature to split on at a node will be based on which choice reduces entropy the most. We compute the weighted average of the entropy across all the children nodes resulting from the split based on a feature, and we compare this value across different possible features. The information gain is the measure of how much entropy was reduced after a given split; this is computed by subtracting the weighted sum of the entropy in all the children nodes from the entropy of the parent node.
The overall process of building a decision tree looks like

1. Start with all examples at the root node

2. Calculate the information gain for all possible features and pick the one with the highest information gain

3. Split the dataset according to the selected feature and create the branches of the tree

4. Keep repeating the process until the stopping criteria is met

One–hot encoding can help us in the case where features have more than two possible values. As opposed to just creating n branches, where n is the amount of possible values the features have, we will create n binary features. For instance, if the feature *colour* can take the values *red, blue, and green*, we will create three features:*red or not red*, *blue or not blue*, and *green or not green*. This allows us to preserve the morphology of a binary tree!
When features can take a continuous value (as opposed to discrete values), a way of working around them on decision trees is to decide a threshold, and classify examples on whether or not they are larger than such threshold. This split is also decided based on which one can give you the highest information gain.
Regression can also be achieved through decision trees; average the continuous value of every element in each of the leaves and this will be your prediction for every other element that ends there. The rationale applied when choosing a split is different for regression trees however: instead of trying to reduce entropy, our goal is to reduce the variance between all the continuous labels of the elements of each subgroup. Similarly to the case of classification through decision trees, we have to compute the weighted sum of the variance of each branch that arises from the split so that we can compare them accurately and subtract this from the variance of the root node.
If we train several decision trees, an *ensemble* of decision trees, we get a much better performance; a single tree is highly sensitive to small changes in the data. The core idea behind the ensemble of trees is having each of them process the input data and output a prediction, so that the model outputs the most "popular" label. Having several trees "vote" instead of just outputting what one says is better because it makes the overall algorithm less sensitive to what a single tree may do.
Sampling with replacement is a technique used to generate all these plausible but slightly different decision trees: we will construct slightly different training sets by continuously picking a random item from the original set until we have a new set of the same size as the original one. The replacement part is key — otherwise, it is just the same elements in different order.
So, given a training set of size m, we will use sampling with replacement to create a new training set of size m and train a decision tree on the new dataset. These two steps must be repeated an n amount of times. Having built the ensemble, you just provide the input to all the trees and have them vote on the correct output label. Sometimes you end up with the same root split feature for a large percentage of the trees; to prevent this, a further randomising elemetn is introduced: at each node, when choosing a feature to use to split, if k features are available, pick a random subset of $j < k$ features and allow the algorithm to only choose from that subset of features. A typical choice of j is $\sqrt{k}$
An even stronger algorithm is **XGBoost**, eXtreme Gradient Boosing. The algorithm behind it is similar to the one for random tree ensembles:

1. Given training set of size m
2. Repeat for B times:
   a) Use sampling with replacement to create a new training set of size m
   Instead of picking from all examples with equal ($\frac{1}{m}$) probability, make it more likely to pick misclassified examples from previously trained trees
   b) Train a decision tree on the new dataset

XGBoost is hard to implement, so most practitioners just do the following:

```
#for classificatioin
from xgboost import XGBClassifier

model = XGBClassifier()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

#for regression
from xgboost import XGBRegressor

model = XGBRegressor()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Decision Trees	**Neural Networks**
Work well on tabular/structured data	Good for all types of data (images, audio, text)
Faster to train	Slower
Small ones are human-interpretable	Works with transfer learning

Lastly, observe how to implement a decision tree from scratch below:

```
import numpy as np
from public_tests import *
from utils import *

#Given hot-encoded dataset
X_train = np.array
    ([[1,1,1],[1,0,1],[1,0,0],[1,0,0],[1,1,1],[0,1,1],[0,0,0],[1,0,1],[0,1,0],[1,0,0]

y_train = np.array([1,1,0,0,1,0,0,1,1,0])

def compute_entropy(y):
    """
    Computes the entropy for

    Args:
       y (ndarray): Numpy array indicating whether each example at a
    node is
```

```
            edible (`1`) or poisonous (`0`)

    Returns:
        entropy (float): Entropy at that node

    """
    entropy = 0.

    if len(y) != 0:
     p1 = len(y[y == 1]) / len(y)
     # For p1 = 0 and 1, set the entropy to 0 (to handle 0log0)
     if p1 != 0 and p1 != 1:
         entropy =  -p1 * np.log2(p1) - (1 - p1) * np.log2(1 - p1)
     else:
         entropy = 0

    return entropy

def split_dataset(X, node_indices, feature):
    """
    Splits the data at the given node into
    left and right branches

    Args:
        X (ndarray):             Data matrix of shape(n_samples, n_features)
        node_indices (list):     List containing the active indices. I.e, the samples being considered at this step.
        feature (int):           Index of feature to split on

    Returns:
        left_indices (list):     Indices with feature value == 1
        right_indices (list):    Indices with feature value == 0
    """

    left_indices = []
    right_indices = []

    for i in node_indices:
       if X[i][feature] == 1:
           left_indices.append(i)
       else:
           right_indices.append(i)
    return left_indices, right_indices

def compute_information_gain(X, y, node_indices, feature):

    """
    Compute the information of splitting the node on a given feature

    Args:
        X (ndarray):            Data matrix of shape(n_samples, n_features)
        y (array like):         list or ndarray with n_samples
```

```
    containing the target variable
        node_indices (ndarray): List containing the active indices.
    I.e, the samples being considered in this step.

    Returns:
        cost (float):        Cost computed

    """
    # Split dataset
    left_indices, right_indices = split_dataset(X, node_indices,
    feature)

    # Some useful variables
    X_node, y_node = X[node_indices], y[node_indices]
    X_left, y_left = X[left_indices], y[left_indices]
    X_right, y_right = X[right_indices], y[right_indices]
    information_gain = 0

    node_entropy = compute_entropy(y_node)
    left_entropy = compute_entropy(y_left)
    right_entropy = compute_entropy(y_right)

    # Weights
    w_left = len(X_left) / len(X_node)
    w_right = len(X_right) / len(X_node)

    #Weighted entropy
    weighted_entropy = w_left * left_entropy + w_right *
    right_entropy

    #Information gain
    information_gain = node_entropy - weighted_entropy

    return information_gain


def get_best_split(X, y, node_indices):
    """
    Returns the optimal feature and threshold value
    to split the node data

    Args:
        X (ndarray):            Data matrix of shape(n_samples,
    n_features)
        y (array like):         list or ndarray with n_samples
    containing the target variable
        node_indices (ndarray): List containing the active indices.
    I.e, the samples being considered in this step.

    Returns:
        best_feature (int):     The index of the best feature to
    split
    """

```

```
    # Some useful variables
    num_features = X.shape[1]
    best_feature = -1
    max_info_gain=0
    for feature in range(num_features):
        info_gain = compute_information_gain(X, y, node_indices, feature)
        if info_gain > max_info_gain:
            max_info_gain = info_gain
            best_feature = feature

    return best_feature


tree = []

def build_tree_recursive(X, y, node_indices, branch_name, max_depth, current_depth):
    """
    Build a tree using the recursive algorithm that split the dataset into 2 subgroups at each node.
    This function just prints the tree.

    Args:
        X (ndarray):            Data matrix of shape(n_samples, n_features)
        y (array like):         list or ndarray with n_samples containing the target variable
        node_indices (ndarray): List containing the active indices. I.e, the samples being considered in this step.
        branch_name (string):   Name of the branch. ['Root', 'Left', 'Right']
        max_depth (int):        Max depth of the resulting tree.
        current_depth (int):    Current depth. Parameter used during recursive call.

    """

    # Maximum depth reached - stop splitting
    if current_depth == max_depth:
        formatting = " "*current_depth + "-"*current_depth
        print(formatting, "%s leaf node with indices" % branch_name, node_indices)
        return

    # Otherwise, get best split and split the data
    # Get the best feature and threshold at this node
    best_feature = get_best_split(X, y, node_indices)

    formatting = "-"*current_depth
    print("%s Depth %d, %s: Split on feature: %d" % (formatting, current_depth, branch_name, best_feature))

    # Split the dataset at the best feature
```

```
156     left_indices, right_indices = split_dataset(X, node_indices,
    best_feature)
157     tree.append((left_indices, right_indices, best_feature))
158 
159     # continue splitting the left and the right child. Increment
    current depth
160     build_tree_recursive(X, y, left_indices, "Left", max_depth,
    current_depth+1)
161     build_tree_recursive(X, y, right_indices, "Right", max_depth,
    current_depth+1)
162 
163 
164 build_tree_recursive(X_train, y_train, root_indices, "Root",
    max_depth=2, current_depth=0)
165 generate_tree_viz(root_indices, y_train, tree)
```