Artificial Neural Networks

Introduction

The concept of Artificial Neural Networks is to have an algorithm that simulates the behaviour of actual mammalian neurons. That makes the study of these algorithms of particular interest in the field of neuroscience. There are many different variations of neural networks, each with its own advantages and disadvantages.

Artificial neural networks have been used for aiding systems to learn. One classic example is image recognition. Artificial neural networks can be utilized to train an algorithm to recognise images. The network is composed of nodes often called artificial neurons, thus mimicking the structure of a biological brain. Each connection can transmit a signal from one node to another node, typically this signal is a real number. The output of each node/artificial neuron is computer using a non-linear function of the sum of the inputs.

Concepts

Artificial neural networks (ANN) are designed to simulate the behavior of actual, biological networks. There are some concepts that form the foundations of ANN's. One of the most fundamental is Hebb's rule. Hebb's rule states that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. Therefore, if two neurons regularly fire simultaneously, then the connection between them will be strengthened. Hebb's rule also indicates that the opposite is also true: if two neurons rarely fire simultaneously, the connection between them will weaken. This has been put more colloquially as "neurons that fire together wire together". This is true for ANN's as well. As two

nodes in an artificial neural network fire together more often, the link between them becomes stronger.

Artificial neural networks consist of nodes that are connected to each other to form a network. The connections between the nodes can be directed and have weights. In ANN's the learning process involves the network itself changing parameters due to feedback from previous iterations. This is typically accomplished by adjusting the weightings between nodes. By changing the weightings between nodes, the likelihood of specific nodes firing in subsequent iterations is altered.

The nodes are organized into layers, the various layers work to process data. Those connections may flow in only one direction, or in both directions. The level of connectivity within an ANN also varies with different types of ANN. The variation of ANN's is substantial.

ANN Terminology

As you delve deeper into artificial neural networks, there is important terminology you should know. A basic glossary is provided here. Additional terminology will be introduced as needed in this tutorial.

Activation Function This function determines if the inputs to a node reach the threshold to cause the node to fire to the node in the next layer. We will examine some specific activation functions later in this section.

Accuracy is a measure the performance of classification, classification with missing inputs, and transcription, accuracy is the proportion of examples for which the model produces the correct output. Accuracy is affected by weights and biases.

Cross Entropy is a metric for estimating how well a model would generalize to new data by testing the model against one or more non-overlapping data subsets withheld from the training set.

Cross Validation is repeated use of the same data, but split differently (i.e., different training and testing sets).

Dimensionality Reduction Dimension reduction summarizes a dataset using its common occurring patterns. Dimension reduction finds patterns in data, and uses these patterns to re-express it in a compressed form. This makes subsequent computation with the data much more efficient.

Error E, a function that computes the inaccuracies of the network as a function of the outputs y and targets t.

Error Rate is a performance metric of the classification of classification, classification with missing inputs, and transcription, the proportion of examples for which the model produces an incorrect output.

Feature measurable property of an object or event with respect to a set of characteristics.

Inputs An input vector is the data given as one input to the algorithm. This will usually be a vector. Written as x, with elements xi, where i runs from 0 to the number of input dimensions, n.

One-Hot this is a group of bits with a high bit 1 and a low bit 0. One hot encoding is used to indicate state in some systems.

Outputs The output vector is y, with elements yj, where j runs from 1 to the number of output dimensions, n. We can write y(x,W) to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights of the network.

Overfitting occurs when comparing the complexity of hypothesis class H with the complexity of the function underlying the data, H is too complex, and the data is not enough to constrain it.

Precision To measure the performance of classification, precision is the fraction of detections reported by the model that were correct.

Stochastic Gradient Descent Also known as incremental gradient descent, it allows one to approximate the gradient with a single data point instead of all available data. At each step of the gradient descent, a randomly chosen data point is used to compute the gradient direction.

Targets The target vector t, with elements tj, where j runs from 1 to the number of output dimensions, n, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

Tensors are primary data structure in TensorFlow programs. Tensors are N-dimensional data structures, most commonly scalars, vectors, or matrices. The elements of a Tensor can hold integer, floating-point, or string values.

Training Set Data is typically divided into a training set and a testing set.

Underfitting occurs when comparing the complexity of hypothesis class H with the complexity of the function underlying the data, H is less complex than the function.

Weights wij, are the weighted connections between nodes i and j. For neural networks these weights are analogous to the synapses in the brain.

Activation Functions

Activation functions, mentioned, earlier in this tutorial, are key to the function of artificial neural networks. An activation function takes the input and determines if the input to the current neuron is sufficient to cause that neuron to fire the next neuron. Activation functions

define the learning pattern and determine the efficiency of the algorithm. This section describes some commonly used activation functions.

Google describes activation functions as "A function (for example, ReLU or sigmoid) that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer."[1]

Rectified Linear Unit (or ReLU) is commonly used in Tensorflow. This function will take the input, and if it is positive will simply output that input, with no changes. If it is not positive, then the ReLU activation function will output 0. The ReLU function is defined by the formula in equation 10.1.

$$\begin{cases} 0 \; if x \leq 0 \\ x \; if x > 0 \end{cases} \qquad \textit{(eq. 10.1)}$$

ReLU is widely used but suffers from something called the dying ReLU problem. Essentially, during the training phase, some nodes/neurons cease functioning or outputting anything other than 0. Essentially, these nodes die. There are cases wherein as much as half the nodes die. One answer to that is the Leaky ReLU. This activation function is essentially a ReLU that has a parameter that determines how much the function leaks. That leakage prevents the death of nodes.

There are other variations of the ReLU function. One often used with Tensorflow is the ReLU6. This activation function has been shown to be faster than traditional ReLU. The ReLU6 function is defined by the formula in equation 10.2.

$$f(x) = \min(\max(0,x),6) \qquad \textit{(eq. 10.2)}$$

---

[1] https://developers.google.com/machine-learning/glossary

The Sigmoid function is also frequently seen in Tensorflow. This takes the input and compresses it into a range between 0 and 1. A result of 0 would mean the current node does not fire, whereas a 1 would mean it is fully firing. The Sigmoid activation function suppresses gradients. This is sometimes called the logistic function and is defined by the formula in equation 10.3.

$$0 - (x) = \frac{1}{1+e^{x}} \qquad \textit{(eq. 10.3)}$$

The sigmoid function was first published in the 1990's. The fact that it suppresses gradients was a substantial advance in activation functions.

The Hyperbolic Tangent (Tanh) activation function is similar to Sigmoid, except that it compresses the input to a range between -1 and +1. This activation function is defined by the formula in equation 10.4.

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \qquad \textit{(eq. 10.5)}$$

Unlike the sigmoid function, the tanh function is zero centered. However, like the sigmoid function, it is computationally expensive due to the exponential operations in the function.

Optimization Algorithms

Optimization algorithms are used to adjust parameters in order to minimize the cost (computational cost) of the function. For each of these algorithms, this section will present a general description. The details of these algorithms are not necessary because Tensorflow handles the mathematical implementation of the algorithm for you. This is not an exhaustive list of all optimization algorithms available in Tensorflow. This description of these key algorithms

should help you understand optimization as a concept. You can view all current Tensorflow optimization algorithms at the Tensorflow website[2].

Gradient descent is an optimization algorithm used to minimize a given function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, gradient descents are useful to update the parameters of the model being used. Parameters refer to coefficients in linear regression and weights in neural networks. Put in more rigorous mathematical terms, the gradient descent algorithm is used to find the minimum of a function. Put more simply, gradient descent is an optimization algorithm used to find the values of parameters of a function that minimizes a cost function (computational cost). When using Tensorflow, the class SGD is the gradient descent optimizer. It should be noted that gradient descent is perhaps the most common optimization algorithm used.

Another optimizer is Adaptive Moment Estimation (ADAM). This is a variation of gradient descent. In fact, ADAM combines two different gradient descent approaches: Root Mean Square Propagation and Adaptive Gradients. Rather than use the entire data set to calculate the gradient, Adaptive Moment Estimation (ADAM) uses a randomly selected subset of the data. This creates a stochastic approximation of the gradient descent. ADAM is also a widely used optimization algorithm.

NADAM is a variation of ADAM that uses a Nesterov Momentum. This, of course, necessitates a discussion of what a Nesterov momentum is. Any gradient descent can be modified with momentum. In this context, momentum is some adjustment to the gradient descent parameter so that movement through the parameter space is averaged over multiple steps.

---

[2] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Normally this is done by introducing velocity. The goal is that momentum will increase in those directions that lead to the most improvement. Nesterov momentum is a variation of that concept of momentum. Rather than calculate momentum with the actual positions in the search space, it calculates based on projected positions in the search space.

Adaptive Gradient (AdaGrad) is actually a group of closely related algorithms. As the name suggests, is a variation of gradient descent. A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable, thus AdaGrad seeks to overcome that limitation. AdaGrad allows step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients observed for the variable.

Models

Models are basically files that are trained to recognize patterns. The model is trained using a set of training data. After training, the model can be used to make predictions from the data. NVIDIA, a graphics card and processor manufacturer, describes models in the following manner: "A machine learning model is an expression of an algorithm that combs through mountains of data to find patterns or make predictions. Fueled by data, machine learning (ML) models are the mathematical engines of artificial intelligence."[3] Tensorflow has an online repository of models that have already been trained for various purposes[4].

Feed Forward Neural Networks

The name of these networks derives from the fact that outputs from nodes in one layer can only go to nodes in the next layer. There is no possibility for a cycle or loopback. The information 'feeds forward', thus the name. This is among the simplest forms of a neural

---

[3] https://blogs.nvidia.com/blog/2021/08/16/what-is-a-machine-learning-model/
[4] https://tfhub.dev/

network. In fact, many of the concepts in the Feed Forward Network are found in all neural networks. Thus, this particular variation bears close study.

The simplest type of feed forward network is the single layer perceptron. The concept of a perceptron predates not only machine learning, but even digital computers. Warren McCulloch and Walter Pitts published an interesting paper in 1943 intitled "A Logical Calculus of Ideas Immanent in Nervous Activity"[5]. Their goal was to simply model how neurons work. This modeling led, eventually, to the creation of artificial neurons. Each neuron takes in input, sums the input, and based on that summation, determines if it will fire or not. The model is shown in figure 10.1

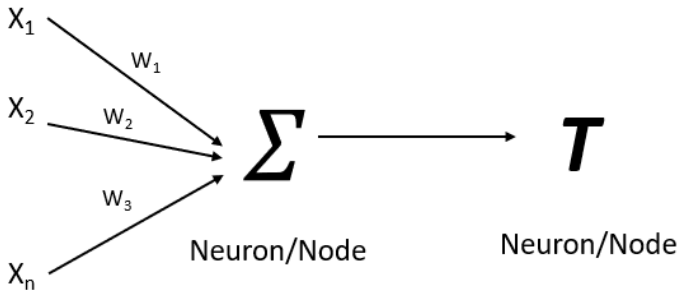

Figure 10.1 McCulloh-Pitts Neuron

Preceding nodes (X_1 through X_n) send input to the node in question. Each of these has a specific weight associated with it (W_1 through W_n). That node sums the input (thus the summation symbol) and if the sum of the weighted input meets or exceeds a specific threshold, then the node in question will fire and send a signal to the next node, the target node (thus the T representing it). The McCulloh Pitts neuron leads us to what is usually called Heb's rule. Hebb's

[5] https://link.springer.com/content/pdf/10.1007/BF02478259.pdf

rule was discussed previously, but as a reminder, consider this quote from another machine learning textbook "Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So, if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away. The idea is that if two neurons both respond to something, then they should be connected."[6]

Perceptron

The perceptron is the most basic implementation of a neural network. It is the implementation of a McCulloch-Pitts neuron. Figure 10.2 shows code for a very simple perceptron in Python.

```
import numpy as np

class Perceptron(object):

    def __init__(self, no_of_inputs, threshold=100, learning_rate=0.01):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.weights = np.zeros(no_of_inputs + 1)

    def predict(self, inputs):
        inputsum = np.dot(inputs, self.weights[1:]) + self.weights[0]
        if inputsum > 0:
          activate = 1
        else:
          activate = 0
        return activate

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                self.weights[1:] += self.learning_rate * (label - prediction) * inputs
                self.weights[0] += self.learning_rate * (label - prediction)
```

Figure 10.2 Perceptron

[6] Machine Learning: An Algorithmic Perspective, Second Edition (Chapman & Hall/Crc Machine Learning & Pattern Recognition)

Just 23 lines of code, including blank spaces. The only problem is when you execute this code it does not print anything out. That is because we did not add any code to print out. We will add that later. For now, notice the three functions in the class Perceptron. We have an initialization function (_init_) that is meant to set initial values. Then we have two functions, one to train and the other to predict. This is a rather simple example, let us look at a more complex example.

The following code uses the mnist dataset. That is the Modified National Institute of Standards and Technology handwriting dataset. This database consists of handwritten numerals. It is often used in machine learning classes to illustrate how to train an algorithm to recognize digits. This dataset has 60,000 training images and 10,000 test images. The data was normalized to fit into 28X28 pixel boundaries.

```
import tensorflow as tf
import tensorflow_datasets as tfds

#the mnist dataset is loaded into two sections
# a test and a training section
(data_train, data_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

def normalize_image(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  return tf.cast(image, tf.float32) / 255., label

data_train = data_train.map(
    normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
data_train = data_train.cache()
data_train = data_train.shuffle(ds_info.splits['train'].num_examples)
data_train = data_train.batch(128)
```

```
data_train = data_train.prefetch(tf.data.AUTOTUNE)


data_test = data_test.map(
    normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
data_test = data_test.batch(128)
data_test = data_test.cache()
data_test = data_test.prefetch(tf.data.AUTOTUNE)

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10)
])
model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

model.fit(
    data_train,
    epochs=6,
    validation_data=data_test,
)
```

When you execute this script a few things will happen. First, the system may give you errors related to whether or not you have a GPU. If you don't the algorithm will still run, just not as efficiently. Then the dataset will be loaded. These initial steps are shown in figure 10.3:

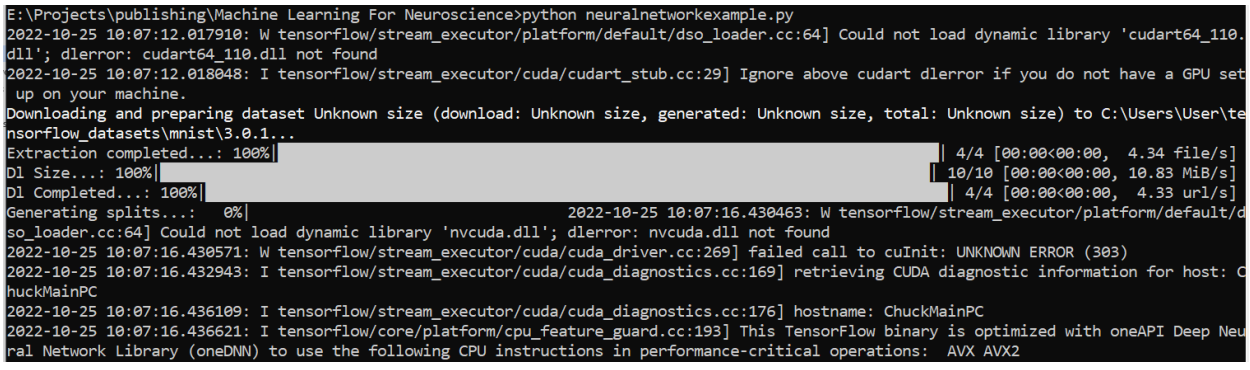

Figure 10.3 First neural network

Next the process will continue through 6 epochs. Notice in the code we set it for 6 epochs. You can experiment with changing that. The output is shown in figure 10.4.

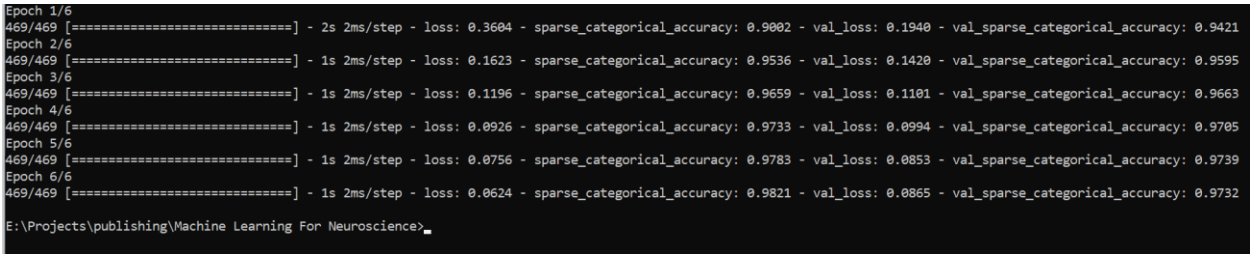

Figure 10.4 First neural network output

This is a more robust example of a neural network. You should pause to get comfortable with the code, and executing the script, before you continue further in this tutoral.

Backpropagation

This is an algorithm used in training artificial neural networks, particularly with feedforward artificial neural networks. Backpropagation is used to compute the gradient of the loss function with respect to the weights. The loss function is also sometimes called a cost or error function. It is a function that is used to represent some cost associated with some activity or event.

The backpropagation algorithm is used to train a neural network using a technique called chain rule. After each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters. Normally the parameters adjusted are the weights and biases.

This algorithm was first introduced in the 1970's but achieved wide recognition due to a 1986 paper in the journal Nature written by David Rumelhart, Geoffrey Hitton, and Ronald Williams. It is useful to consider how these authors described backpropagation:

> "The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure"[7]

Another description from the International Dictionary of Artificial Intelligence[8] could be useful in aiding your understanding of backpropagation:

"A classical method for error propagation when training Artificial Neural Networks (ANNs). For standard backpropagation, the parameters of each node are changed according to the local error gradient. The method can be very slow to converge although it can be improved through the use of methods that slow the error propagation and by batch processing. Many alternate methods such as the conjugate gradient and Levenberg-Marquardt algorithms are more effective and reliable."

Fortunately, the backpropagation algorithm is taken care of by libraries such as TensorFlow. You don't have to implement the details yourself. I say this is fortunate, because this algorithm uses partial derivatives which may be beyond some readers mathematical skillset.

Normalization

---

[7] https://www.nature.com/articles/323533a0

[8] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.375.8194&rep=rep1&type=pdf

Normalization is a data preparation technique that entails altering the values of numeric values in a dataset to a common scale. This is often needed when the values of the dataset have different ranges. There are three different types of normalization used in machine learning:

Input normalization: Normalizing input is quite common. A common example is scaling the pixel values of images (0–255) to values between zero and one.

Batch normalization: This is a normalization that occurs between each layer of the network so that values mean is zero and their standard deviation is one.

Internal normalization: This normalization is about ensuring that each layer keeps the previous layer's mean and variance.

Many neural network code samples you will see implement normalization at some point. Almost all machine learning scripts dealing with image data will at least implement input normalization.

Specific Variations of Neural Networks

Artificial Neural Networks are widely used, perhaps the most widely used machine learning algorithms. Due to that fact, it should come as no surprise that there are numerous variations of these algorithms. Each has some advantage for some specific application. This section will briefly describe major variations.

Recurrent Neural Networks

The recurrent neural network (RNN) is a common variation of neural networks. These are derived from feedforward neural networks. Recurrent nets are a variation of artificial neural network intended to distinguish patterns in sequences of data. The patterns can be almost anything, but the pattern recognition makes RNN’s idea for identifying handwriting, numbers,

time series data from sensors, stock market data, and more. RNN's consider time and sequence. This gives these algorithms a temporal dimension.

The manner in which RNNs function is to have cycles that permit output from some nodes to affect subsequent input into the same nodes. There are several types of RNN's including:

- Encoder decoder or sequence to sequence RNNs,
- Bidirectional RNNs,
- Recursive RNNs,
- Gated Recurrent Unit (GRU),
- LSTM RNNs.

The following description of RNN's may help clarify this class of algorithms for you "A family of neural networks for processing sequential data. RNNs share the same weights across different discrete time steps: each member of the output is a function of the previous members of the output; each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph. RNNs are very powerful dynamic systems for tasks that involve sequential inputs, such as speech and language."[9]

Convolutional Neural Networks

A convolutional neural network (CNN) is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that do convolutional operations. The concept is shown in figure 10.5.

---

[9] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. Deep Learning (p. 373). MIT Press.
LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey. Deep Learning. Nature 521, 436-444 (2015)

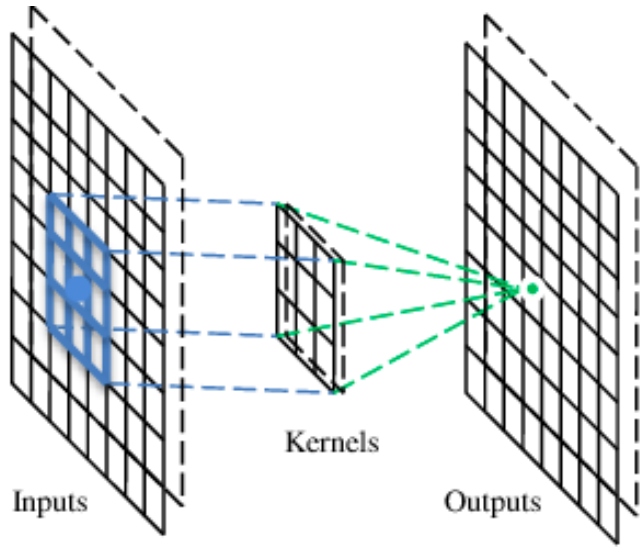

Figure 10.5 Convolutional Neural Network

The different layers perform different aspects of the learning. Convolutional Neural Networks (sometimes called ConvNets) are often used with visual imagery. This makes them an ideal candidate for examining diagnostic imagery. Put more formally, convolutional neural networks use a mathematical operation called convolution rather than basic matrix multiplication in at least one of the layers. The architecture of a convolutional neural network will include an input layer, one or more hidden layers, and an output layer. The input is provided as a tensor with a shape. The convolutional neural layer transforms that image to a feature map with a shape. Some sources refer to the feature map as an activation map. The convolutional layers each respond to input only for their own receptive field. This is quite similar to how the brain processes visual imagery.

The following code should help elucidate the convolutional neural network. You may also wish to compare this to the previous code examples, noting similarities and differences.

```
#! /usr/bin/env python
# This is a basic convolutional neural network example.

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

```
import tensorflow as tf

#tensorflow includes a 2D convolutional network you just
#have to include it.
from tensorflow.keras.layers import Dense, Flatten, Conv2D
from tensorflow.keras import Model


# Load and prepare the MNIST dataset. This dataset is incorporated with tensorflow
# the data is split into a training set and a test set
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# # Add a channels dimension
x_train = x_train[..., tf.newaxis]
x_test  = x_test[..., tf.newaxis]

# Use tf.data to batch and shuffle the dataset
train_ds = tf.data.Dataset.from_tensor_slices(
       (x_train, y_train)).shuffle(10000).batch(32)
test_ds = tf.data.Dataset.from_tensor_slices(
       (x_test, y_test)).batch(32)

# Build the tf.keras model using the Keras model the layers will use
# different activation functions
class MyModel(Model):
  def __init__(self):
     super(MyModel, self).__init__()
     self.conv1 = Conv2D(32, 3, activation='relu')
     self.flatten = Flatten()
     self.d1 = Dense(128, activation='relu')
     self.d2 = Dense(10, activation='softmax')

  def call(self, x):
     x = self.conv1(x)
     x = self.flatten(x)
     x = self.d1(x)
     return self.d2(x)

# Create an instance of the model
model = MyModel()

# Choose an optimizer and loss function for training
# there are several other optimizers you can expiriment with
```

```
# SGD,RMSprop, Adam,  Adamax, etc. To see a complet list go to
# https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

# Select metrics to measure the loss and the accuracy of the model
# a complete list of metrix can be found here
https://www.tensorflow.org/api_docs/python/tf/keras/metrics
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')

# Use tf.GradientTape to train the model.
@tf.function
def train_step(images, labels):
   with tf.GradientTape() as tape:
      predictions = model(images)
      loss = loss_object(labels, predictions)
   gradients = tape.gradient(loss, model.trainable_variables)
   optimizer.apply_gradients(zip(gradients, model.trainable_variables))
   train_loss(loss)
   train_accuracy(labels, predictions)

@tf.function
def test_step(images, labels):
   predictions = model(images)
   t_loss = loss_object(labels, predictions)
   test_loss(t_loss)
   test_accuracy(labels, predictions)

# you may wish to change the number of epochs to see
# what effect that has
EPOCHS = 5

for epoch in range(EPOCHS):
   for images, labels in train_ds:
      train_step(images, labels)

   for test_images, test_labels in test_ds:
      test_step(test_images, test_labels)

   template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
   print(template.format(epoch+1,
                         train_loss.result(),
```

```
                    train_accuracy.result()*100,
                    test_loss.result(),
                    test_accuracy.result()*100))
  # Reset  metrics for the next epoch
  train_loss.reset_states()
  train_accuracy.reset_states()
  test_loss.reset_states()
  test_accuracy.reset_states()
```

Much of the preceding code should be familiar to you after the previous code samples. A few elements may require some further explanation. One such item is SparseCategoricalCrossentropy. This is a property of Keras that is used when there are two or more label classes, in order to compute the crossentropy loss between the labels and predictions. When you execute this script you may first get familiar GPU error messages if your computer does not have a suitable GPU. Then you will see the epochs each with an accuracy shown. This is shown in figure 10.6.

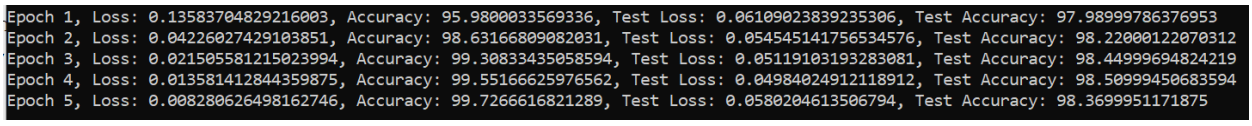

Figure 10.7 CNN Output

Note that the test accuracy improved to a point, then began to level off and even decrease slightly. This is an important fact to keep in mind. The issue is not simply to do as many epochs as you possibly can. There is a point were this will simply generate diminishing returns. However, experimenting with different activation and optimization functions can be quite useful.

Given how common convolutional neural networks are, we will review a second code sample. This one is used to detect pituitary brain tumors based on imaging. The code is more detailed and should aid you in understanding CNN's. The data can be found on Github at https://github.com/sartajbhuvaji/brain-tumor-classification-dataset

```
import numpy as np
import pandas as pd # used for reading CSV data files

import os #needed for navigating file system

#The path is based on having downloaded the data to a
#subfolder of the directory this script is in and the
#folder was named 'braintumordataset'
for dirname, _, filenames in os.walk('/BrainTumorDataSet'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

#keras is used for our CNN
import keras
from keras.models import Sequential
from keras.layers import Conv2D,Flatten,Dense,MaxPooling2D,Dropout
from sklearn.metrics import accuracy_score


import io
from PIL import Image
import tqdm
from sklearn.model_selection import train_test_split
import cv2
from sklearn.utils import shuffle
import tensorflow as tf

import matplotlib.pyplot as plt

xtrainarray = []
ytrainarray = []
imgsize = 150
labels = ['glioma_tumor','meningioma_tumor','no_tumor','pituitary_tumor']
for i in labels:
    folderPath = os.path.join('BrainTumorDataSet/Training/',i)
    for j in os.listdir(folderPath):
        img = cv2.imread(os.path.join(folderPath,j))
        img = cv2.resize(img,(imgsize,imgsize))
        xtrainarray.append(img)
        ytrainarray.append(i)

for i in labels:
    folderPath = os.path.join('BrainTumorDataSet/Testing/',i)
    for j in os.listdir(folderPath):
        img = cv2.imread(os.path.join(folderPath,j))
        img = cv2.resize(img,(imgsize,imgsize))
```

```
        xtrainarray.append(img)
        ytrainarray.append(i)

xtrainarray = np.array(xtrainarray)
ytrainarray = np.array(ytrainarray)

xtrainarray,ytrainarray = shuffle(xtrainarray,ytrainarray,random_state=101)
xtrainarray.shape

xtrainarray,X_test,ytrainarray,y_test =
train_test_split(xtrainarray,ytrainarray,test_size=0.1,random_state=101)

ytrainarray_new = []
for i in ytrainarray:
    ytrainarray_new.append(labels.index(i))
ytrainarray=ytrainarray_new
ytrainarray = tf.keras.utils.to_categorical(ytrainarray)

y_test_new = []
for i in y_test:
    y_test_new.append(labels.index(i))
y_test=y_test_new
y_test = tf.keras.utils.to_categorical(y_test)

# you should expiriment with different activation functions
# Tanh, relu, relu6, gelu, etc.
model = Sequential()
model.add(Conv2D(32,(3,3),activation = ‘relu6’,input_shape=(150,150,3)))
model.add(Conv2D(64,(3,3),activation= ‘relu6’))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.3))
model.add(Conv2D(64,(3,3),activation=‘relu6’))
model.add(Conv2D(64,(3,3),activation=‘relu6’))
model.add(Dropout(0.3))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.3))
model.add(Conv2D(128,(3,3),activation=‘relu6’))
model.add(Conv2D(128,(3,3),activation=‘relu6’))
model.add(Conv2D(128,(3,3),activation=‘relu6’))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.3))
model.add(Conv2D(128,(3,3),activation='relu'))
model.add(Conv2D(256,(3,3),activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Dropout(0.3))
model.add(Flatten())
```

```
model.add(Dense(512,activation = 'relu'))
model.add(Dense(512,activation = 'relu'))
model.add(Dropout(0.3))
model.add(Dense(4,activation='softmax'))

model.summary()


model.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])
#5 epochs is just so this will run quickly. In real applications you will often
# use more.
history = model.fit(xtrainarray,ytrainarray,epochs=5,validation_split=0.1)

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(len(acc))
fig = plt.figure(figsize=(14,7))
plt.plot(epochs,acc,'r',label="Training Accuracy")
plt.plot(epochs,val_acc,'b',label="Validation Accuracy")
plt.legend(loc='upper left')
plt.show()


loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(loss))
fig = plt.figure(figsize=(14,7))
plt.plot(epochs,loss,'r',label="Training loss")
plt.plot(epochs,val_loss,'b',label="Validation loss")
plt.legend(loc='upper left')
plt.show()

img = cv2.imread('/BrainTumorDataSet/Training/pituitary_tumor/p (107).jpg')
img = cv2.resize(img,(150,150))
img_array = np.array(img)
img_array.shape

img_array = img_array.reshape(1,150,150,3)
img_array.shape


from tensorflow.keras.preprocessing import image
img = image.load_img('/BrainTumorDataSet/Training/pituitary_tumor/p (107).jpg')
plt.imshow(img,interpolation='nearest')
plt.show()
```

```
a=model.predict(img_array)
indices = a.argmax()
indices
```

Notice the line of code model.summary(), model.summary will display the choices you made in your model on the screen. You could omit this line and the script would still work. Or you may try including this line in other scripts where you build a keras.model. When you execute this script, several images will be displayed, these are shown in figures 10.7 through 10.10.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896

conv2d_1 (Conv2D)            (None, 146, 146, 64)      18496

max_pooling2d (MaxPooling2D  (None, 73, 73, 64)        0
)

dropout (Dropout)            (None, 73, 73, 64)        0

conv2d_2 (Conv2D)            (None, 71, 71, 64)        36928

conv2d_3 (Conv2D)            (None, 69, 69, 64)        36928

dropout_1 (Dropout)          (None, 69, 69, 64)        0

max_pooling2d_1 (MaxPooling  (None, 34, 34, 64)        0
2D)

dropout_2 (Dropout)          (None, 34, 34, 64)        0

conv2d_4 (Conv2D)            (None, 32, 32, 128)       73856

conv2d_5 (Conv2D)            (None, 30, 30, 128)       147584

conv2d_6 (Conv2D)            (None, 28, 28, 128)       147584

max_pooling2d_2 (MaxPooling  (None, 14, 14, 128)       0
2D)
```

Figure 10.7 CNN Model Summary

```
Epoch 1/5
83/83 [==============================] - 98s 1s/step - loss: 1.4175 - accuracy: 0.2838 - val_loss: 1.3652 - val_accuracy: 0.2891
Epoch 2/5
83/83 [==============================] - 98s 1s/step - loss: 1.3617 - accuracy: 0.2762 - val_loss: 1.3630 - val_accuracy: 0.2721
Epoch 3/5
83/83 [==============================] - 103s 1s/step - loss: 1.3593 - accuracy: 0.2872 - val_loss: 1.3633 - val_accuracy: 0.2721
Epoch 4/5
83/83 [==============================] - 99s 1s/step - loss: 1.3612 - accuracy: 0.2720 - val_loss: 1.3622 - val_accuracy: 0.2891
Epoch 5/5
83/83 [==============================] - 101s 1s/step - loss: 1.3597 - accuracy: 0.2788 - val_loss: 1.3648 - val_accuracy: 0.2721
```

Figure 10.8 CNN Epoch Results

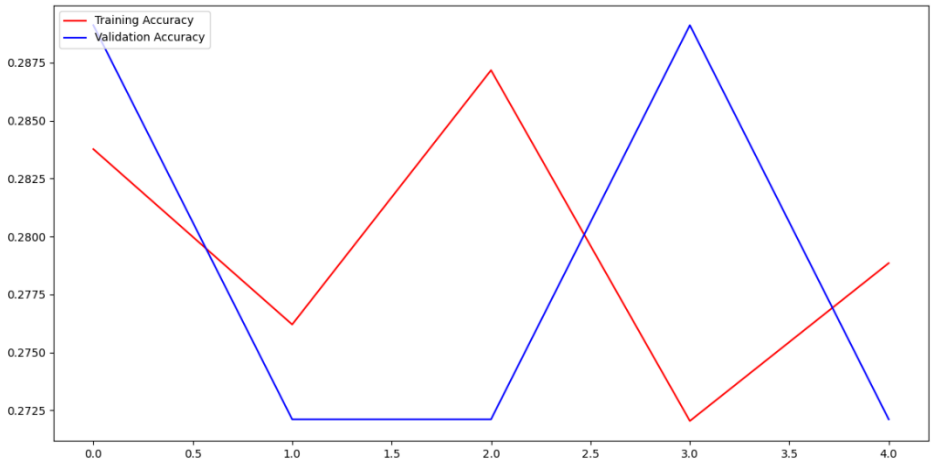

Figure 10.9 CNN Accuracy

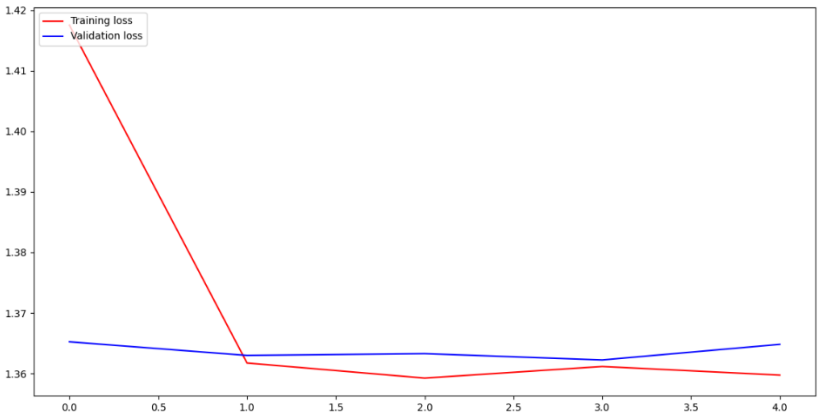

Figure 10.10 CNN Loss

Autoencoder

This is a variation of artificial neural network that is used to learn how to properly code unlabeled data. The algorithm validates its results by trying to regenerate the input data from the encoders. The encoding is simply a representation of the data. Autoencoders are often used for dimensionality reduction. An autoencoder has two main parts: an encoder that maps the message to a code, and a decoder that reconstructs the message from the code. An optimal autoencoder would perform as close to perfect reconstruction as possible. How close to perfection the autoencoder performs in actuality is determined by a quality function.

As you can probably guess, there are numerous variations of autoencoders. There are sparse autoencoders (SAE), denoising autoencoders (DAE), and others. Sparse autoencoders intentionally introduce bottlenecks, but rather than do so via reducing the number of nodes, the SAE instead penalizes particular activations of nodes with a layer. Denoising autoencoders work by introducing a minor deviation into the input data, but maintaining the unmodified data as the target output. The model needs to learn to denoise (remove deviations) from the data.

The following code is a basic autoencoder. This code again uses the MINST database because of its easy availability.

```
from keras.layers import Dense,Conv2D,MaxPooling2D,UpSampling2D
from keras import Input, Model
from keras.datasets import mnist
import numpy as nump
import matplotlib.pyplot as plt

#first build the model
encoding_dimension = 15
inumput_img = Input(shape=(784,))
# encoded representation of input
encoded = Dense(encoding_dimension, activation='relu')(input_img)
# decoded representation of code
```

```
decoded = Dense(784, activation='sigmoid')(encoded)
# Model which take input image and shows decoded images
autoencoder = Model(input_img, decoded)

# Now build the encoder and decoder
encoder = Model(input_img, encoded)
# Creating a decoder model
encoded_input = Input(shape=(encoding_dimension,))
# last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))

#compile the model. Note the optimizer we are using
#you can try this with a different optimizer.
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
#now we load the minst data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), nump.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), nump.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

When executed, it will simply output the following:

(60000, 784)

(10000, 784)

The following quote might help elucidate the autoencoder “The combination of an encoder function that converts the input data into a different representation, and a decoder function that converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and

then the decoder, but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties."[10]

Spiking Neural Network

Another variation of the artificial neural network is one that incorporates the concept of time. With a spiking neural network information is not transmitted at every propagation cycle. Rather information is only transmitted when a particular threshold is met. This mimics the manner in which biological neurons function. Spiking neural networks are based on the Hodgkin-Huxley model of biological networks. This model describes how action potentials are initiated and propagated to the next neuron. The development of this model earned Alan Hodgkin and Andrew Huxley the 1963 Nobel Prize in Physiology or Medicine.

The concept is for the nodes in a given layer to not test for activation in every iteration of propagation. The nodes only test for activation only when their potentials reach a specific value. This is different than typical multi-layer perceptrons wherein neurons test for propagation every iteration.

Deep Neural Networks

While the terminology may indicate that this is a substantial variation on the neural network concept, it is not. A deep neural network (DNN) simply has more hidden layers. This is shown in figure 10.11.

---

[10] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. Deep Learning (p. 8). MIT Press.

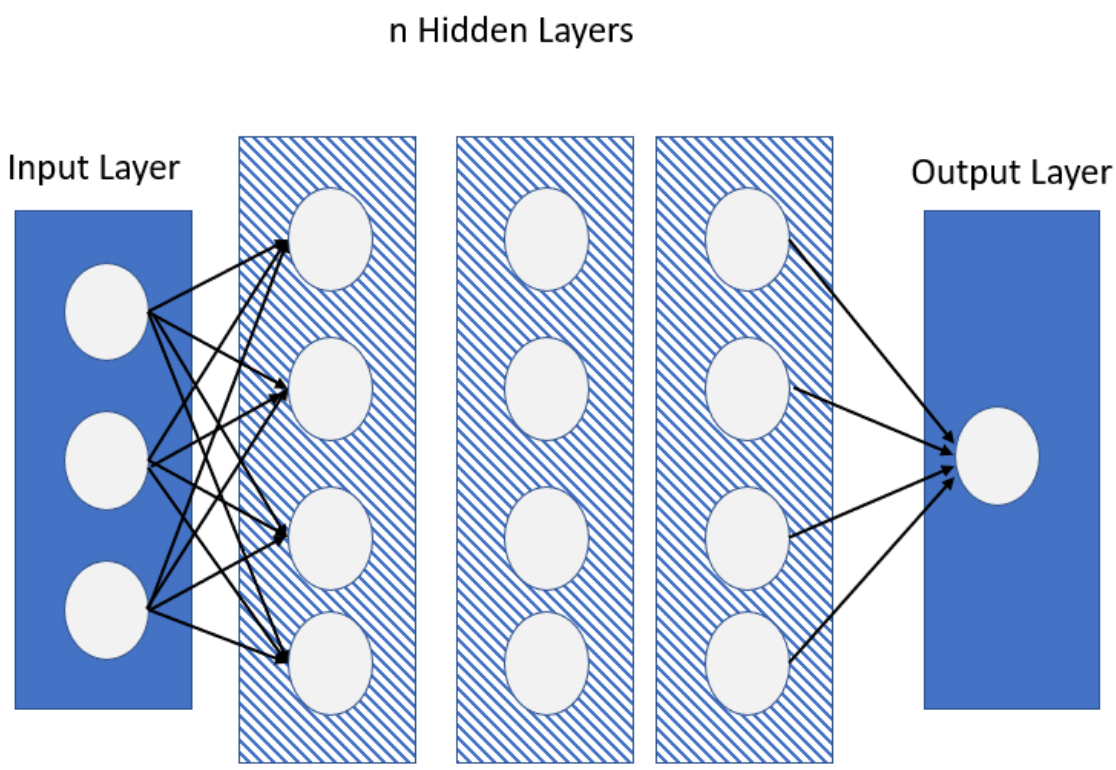

Figure 10.11 Deep Neural Network

In figure 10.11 the connections in the hidden layers are not shown. This is because the hidden layers may have all nodes connected to all other nodes, or only some nodes connected to other nodes. A deep neural network can also be a feed forward network or a recurrent network. Each layer may even use a different activation function. Deciding how many layers and what activation functions should be used is not a straight forward process. It requires some level of experience, and a bit of experimentation to find the right combination for a given problem.

Neuroscience Example Code

The following is an example of Python code to recognize brain tumors from diagnostic imaging data. You will need to do a few things to make this code function. First, if you have not previously done so, you will need to install two items:

pip install opencv-python

pip install scikit-learn

The first allows you to open comma delimited files. The second allows you to use the scikit learn package. Then you need to download the dataset. There are a number of datasets freely available on websites like Kaggle.com. This dataset comes from

https://www.kaggle.com/datasets/preetviradiya/brian-tumor-dataset

You will download it to the same folder you have your Python script in. The code is shown here:

```
import cv2
import os
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
from tqdm import tqdm
from sklearn.model_selection import train_test_split
import numpy as np

# there are two data groups in the set, health and tumor
loc0 = 'Brain Tumor Data Set/Healthy'
loc1 = 'Brain Tumor Data Set/Brain Tumor'

# setup an array for your features
features = []

#tqdm is used to create a progress bar, definitely makes your script more user friendly
for img in tqdm(os.listdir(loc0)):
    f = cv2.imread(os.path.join(loc0,img)) #read in healthy data
    f = cv2.cvtColor(f,cv2.COLOR_BGR2GRAY)
    f = cv2.resize(f,(70,70))
    features.append(f)

for img in tqdm(os.listdir(loc1)):
    f = cv2.imread(os.path.join(loc1,img)) # read in tumor data
    f = cv2.cvtColor(f,cv2.COLOR_BGR2GRAY)
    f = cv2.resize(f,(70,70))
    features.append(f)

#now we have an array for the labels.
labels = []

for img in tqdm(os.listdir(loc0)):
    labels.append(0)
```

```
for img in tqdm(os.listdir(loc1)):
    labels.append(1)


X = np.array(features)
X.shape

Y = np.array(labels)
Y.shape

Xt = X.reshape(4600, 4900)
Yt = Y.reshape(4600,1)


xtrain,xtest,ytrain,ytest = train_test_split(Xt,Yt,train_size=0.75)


ytrainC = tf.keras.utils.to_categorical(ytrain)
ytestC = tf.keras.utils.to_categorical(ytest)

# we are using sequential models with specific activation functions. Once you are comfortable
# with this script you can experiment with different models and different activation functions
model = models.Sequential()

model.add(layers.Dense(300, activation='relu' , input_dim = xtrain.shape[1]))

model.add(layers.Dense(200,activation='relu'))

model.add(layers.Dense(200,activation='relu'))

model.add(layers.Dense(100,activation='relu'))

model.add(layers.Dense(2,activation='sigmoid'))


xtrainN = xtrain/xtrain.max()
xtestN = xtest/xtest.max()

SGD = tf.keras.optimizers.SGD(0.1)
model.compile(loss = 'categorical_crossentropy',
              optimizer = SGD,
              metrics=['Accuracy'])

model.fit(xtrainN,ytrainC, epochs=15, validation_data=(xtestN,ytestC))
```

When you execute the code you should see something like what is shown in figure 10.11.

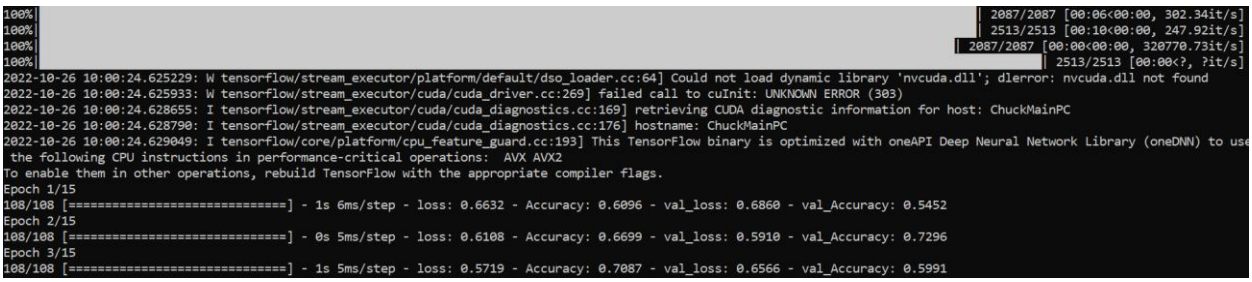

Figure 10.12 Using Brain Tumor Imagery

Note the progress bars as data is being read into the script. Then you see the epochs begin with each having a particular accuracy.

Exercises

Lab 1: Basic TensorFlow

```
#!/usr/bin/python
# this is a basic tensor flow project
import tensorflow as tf

mnist = tf.keras.datasets.mnist

# load the data set
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0


#Build the tensorflow tf.keras.Sequential model by stacking layers.

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

```
#For each example the model returns a vector of "logits"
#or "log-odds" scores, one for each class.

predictions = model(x_train[:1]).numpy()
predictions



#The tensorflow tf.nn.softmax function converts these logits
#to "probabilities" for each class:
tf.nn.softmax(predictions).numpy()


#The losses.SparseCategoricalCrossentropy loss takes a vector of logits
#and a True index and returns a scalar loss for each example.
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
loss_fn(y_train[:1], predictions).numpy()

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)


#The Model.evaluate method checks the models performance

model.evaluate(x_test,  y_test, verbose=2)
```

Now run it

```
D:\MLPython>python firstexample.py
2020-08-05 15:40:04.524759: W tensorflow/stream_executor/platform/default/dso_loader.cc:55] Could not load dynamic libr
ary 'cudart64_101.dll'; dlerror: cudart64_101.dll not found
2020-08-05 15:40:04.561741: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do
not have a GPU set up on your machine.
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
2020-08-05 15:40:31.075274: W tensorflow/stream_executor/platform/default/dso_loader.cc:55] Could not load dynamic libr
ary 'nvcuda.dll'; dlerror: nvcuda.dll not found
2020-08-05 15:40:31.079931: E tensorflow/stream_executor/cuda/cuda_driver.cc:313] failed call to cuInit: UNKNOWN ERROR
(303)
2020-08-05 15:40:31.084328: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic infor
mation for host: WIN-7EP9LVQV3O7
2020-08-05 15:40:31.086703: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: WIN-7EP9LVQV3O7
2020-08-05 15:40:31.097392: I tensorflow/core/platform/cpu_feature_guard.cc:143] Your CPU supports instructions that th
is TensorFlow binary was not compiled to use: AVX2
2020-08-05 15:40:31.291817: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x2be43b16a80 initialized for
 platform Host (this does not guarantee that XLA will be used). Devices:
2020-08-05 15:40:31.296403: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Defaul
```

```
Epoch 1/5
1875/1875 [==============================] - 2s 864us/step - loss: 0.2973 - accuracy: 0.9132
Epoch 2/5
1875/1875 [==============================] - 1s 782us/step - loss: 0.1461 - accuracy: 0.9565
```

Lab 2: Perceptron

```
import numpy as np

X = np.array([
    [-2,4,-1],
    [4,1,-1],
    [1, 6, -1],
    [2, 4, -1],
    [6, 2, -1],

])

y = np.array([-1,-1,1,1,1])

def perceptron_sgd(X, Y):
    w = np.zeros(len(X[0]))
    eta = 1
    epochs = 20

    for t in range(epochs):
        for i, x in enumerate(X):
            if (np.dot(X[i], w)*Y[i]) <= 0:
                w = w + eta*X[i]*Y[i]

    return w

w = perceptron_sgd(X,y)
print(w)
```