PharmaSUG 2024 - Paper RW-390

Unraveling the Layers within Neural Networks: Designing Artificial and Convolutional Neural Networks for Classification and Regression Tasks Using Python's Keras & TensorFlow

Ryan Paul Lafler, Premier Analytics Consulting, LLC;
Anna Wade, Premier Analytics Consulting, LLC

Abstract

Capable of accepting and mapping complex relationships hidden within structured and unstructured data, Neural Networks are composed from layers of neurons with functions that interact, preserve, and exchange information between each other to develop highly flexible and robust predictive models. Neural Networks are versatile in their applications to real-world problems; capable of regression, classification, and generating entirely new data from existing data sources, Neural Networks are accelerating the breakthroughs in deep learning methodologies.

Given the recent advancements in graphical processing unit (GPU) cards, cloud computing, and the availability of interpretable APIs like the Keras interface for TensorFlow, Neural Networks are quickly moving from development to deployment in industries ranging from finance, healthcare, climatology, movies, video streaming, business analytics, and marketing given their versatility in modeling complex problems using structured, semi-structured, and unstructured data. This Paper provides users with an intuitive, example-oriented guide on designing fundamental Artificial Neural Network (ANN) and Convolutional Neural Network (CNN) architectures with Python's Keras and TensorFlow libraries for non-parametric regression and image classification tasks.

1. Introduction

Neural Networks are a non-parametric modeling method that permits the mapping of complex relationships hidden in structured, semi-structured, and unstructured data. The two major types of Network architectures, Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs), represent different methods for uncovering relationships in small-sized, moderate-sized, and big data.

Artificial Neural Networks are composed of several layers of weights and functions that transform inputs to outputs based on iterations of learning for structured and semi-structured tabular data. Convolutional Neural Networks are adept at learning to recognize patterns in unstructured datasets including images, videos, text, and audio sources for regression, classification, and generative tasks.

This Paper introduces important concepts discussing the inner workings of Neural Networks, programming Artificial Neural Networks and Convolutional Neural Networks with TensorFlow and developing Network architectures for non-parametric regression and binary image classification using TensorFlow's Keras API.

Understanding the Importance of Neural Networks and Deep Learning in Industry and Research

Neural Networks are a subset of deep learning that can process, learn, and map hierarchical representations within data. This hierarchical, feed-forward flow allows Networks to learn intricate patterns and features within datasets and map complex relationships between them. Neural Networks, like their machine learning counterparts, encourage the automation of complex, redundant tasks, and as such, has fundamentally changed the way businesses and organizations understand and interact with all types of data in the world.

In TensorFlow, Neural Networks are optimized for training on graphical processing units, commonly abbreviated as GPU cards. The development and release of new GPU cards from NVIDIA, AMD, and Intel has powered modeling breakthroughs in fields containing significant quantities of unstructured data including image recognition, natural language processing, video streaming, video upscaling, time series forecasting, and text analysis.

Convolutional Neural Networks (CNNs) for Computer Vision Tasks

Convolutional Neural Networks (CNNs) are commonly used for pattern recognition in unstructured data. CNNs prove exceptionally useful in automating supervised, semi-supervised, and unsupervised tasks like image recognition, image classification, image generation, object detection, and image segmentation.

Densely connected Neural Networks, typically called Artificial Neural Networks (ANNs), exhibit reduced performance and efficiency when applied to computer vision tasks. ANNs require that images be first flattened to vectors and then passed through layers of neurons, eliminating any information present in the spatial relationships between pixels. CNNs, on the other hand, take advantage of this spatial information by preserving the original dimensions of an image array and passing it through layers that transform and extract relevant features (patterns) *without* needing to be flattened as a vector. The output from each layer is then passed downstream to subsequent layers for additional feature extraction.

Much like Artificial Neural Networks, Convolutional Neural Networks are similarly inspired by nature, borrowing from processing mechanisms attributed to human visual systems. While a person's visual recognition is generated by light and stimuli passing signals to the brain, CNNs deploy kernels in convolutional layers that scan input vectors and arrays to detect patterns, edges, and other relevant features across all regions of the data. These convolutional layers reshape and transform the data as it travels further down the layers of the Network, learning new features by decomposing the original array into smaller and finer-detailed representations of the data.

2. Neural Network Fundamentals

Simplifying the Inner Workings of Neural Networks

Neural Networks are best viewed as composite functions. In a simple hierarchical Network, output from one layer is passed as the input to the next immediate layer. Each layer performs some non-linear transformation on the data, and the multiple layers connecting the Network's input to its output are referred to as its **hidden layers**.

Layers consist of neurons that form edges with neurons in the next immediate layer—edges between neurons are called *connections* and contain the Network's trainable weight parameters. These weights are iteratively updated in batches, permitting the Network to select optimal combinations of weights that minimize some target loss (cost) function.

Since there is almost always more than one weight per layer, each weight has its own partial derivative with respect to the loss function. A first-order derivative measures the rate of change of the loss function when adjusting that weight parameter by a small amount. Collectively, these first-order partial derivatives are stored in a vector called the *gradient*, containing derivatives for all weights inside of the Neural Network. *The gradient measures the loss function's steepest rate of change with respect to all trainable weights inside of the Network*.

A Network's number of weight parameters can grow to include thousands, hundreds of thousands, millions, tens of millions, and even hundreds of millions of trainable parameters, necessitating an efficient approach for computing weights from the gradient that best minimize the loss function.

The first-order partial derivatives for connections between the second-to-last layer and the last layer (the Network's output layer) are the first to be calculated—subsequent partial derivatives are then calculated by moving backwards through connections between the remaining layers. But why move backwards from the output layer instead of forwards from the input layer?

Networks are composite functions, best exemplified as a Russian doll set, where inner functions are stored inside of their outer functions. Furthermore, according to the chain rule for differentiation (first-order derivatives of functions), the last layer's weights serve as an input to all other layers, meaning it should be differentiated first.

This process is called *backpropagation*: where differentiation begins at the output layer (inner function), moving backward through the hidden layers until reaching the Network's input layer (outer function). This allows the algorithm to determine how much each weight contributes to the Network's overall loss—and which weights need to be adjusted to best reduce the overall loss.

Loss Functions and Gradient Descent

Feedback is essential for Neural Networks to improve their performance. Loss functions provide a way to quantify this feedback, usually taking the difference between the observed value and its predicted counterpart. This difference is referred to as the *pseudo-residual*, borrowing from the residual calculated from the difference between observed and predicted values during ordinary least squares (OLS) regression. Common loss functions for regression prediction include Mean Squared Error (MSE), Mean Absolute Error (MAE), and Huber loss.

Once the gradient is calculated from backpropagation, Networks need a process for estimating their weight values that best minimize the loss function during each batch of training. Closed-form solutions for calculating optimal weight parameters (i.e.,

linear regression using the Mean Squared Error loss function) do not exist for the complex, non-linear relationships modeled by Networks. Instead, Networks require a process that approximates locally optimized weight parameters after training on each batch of data over time.

Called *gradient* descent, this method of weight estimation iteratively adjusts the Network to minimize some loss function over several training cycles, referred to as *epochs*. The goal, over enough training epochs, is for gradient descent to estimate optimal combinations of weights within all layers of the Network. There are three forms of gradient descent contingent on the size of the training batches, with each variant determined by the number of training instances comprising each batch.

- **Stochastic Gradient Descent (SGD)** calculates optimal weight parameters after a single instance is passed through the Network. Each batch consists of 1 training instance.
- **Mini-Batch Gradient Descent** calculates optimal weight parameters after some subset of training instances (i.e., 32 instances, 64 instances, 128 instances) smaller than the original dataset is passed through the Network. A subset of training instances is referred to as a *mini-batch*.
- **Batch (Full-Batch) Gradient Descent** passes the entire set of training instances through the Network and then updates weight parameters that best minimize the loss function. There is only one batch, and it contains all the training instances.

Determining the optimal gradient descent method to choose depends on the size of the data, complexity of the loss function, and overall architecture of the Neural Network. Choosing one method over another will lead to differences in how the Network trains—for instance, evaluating the Network after one training instance (the SGD approach) will lead to significantly more variability in training metrics than those produced from batch gradient descent.

Flexibility of Neural Networks for Modeling Structured, Semi-Structured, and Unstructured Data

Given their versatility in accepting inputs from vectors (columns) to multi-dimensional arrays, Neural Networks opened new approaches to mapping complex relationships between tabular, image, video, audio, and text-based datasets.

Traditional machine learning (ML) algorithms such as decision trees and support vector machines require intensive preprocessing and domain expertise for retaining important features within unstructured data. Using these ML algorithms, engineers were unable to pass entire arrays as inputs, and instead, were forced to first vectorize (*flatten*) and extract relevant features from images, videos, and text for classification and regression purposes.

Neural Networks, on the other hand, can accept a multitude of different data structures as inputs to a Network in addition to outputting classification scores and regression predictions for two different tasks at the same time. These Networks, termed multi-input and multi-output Networks, respectively, greatly enhance the preservation of information and tolerance towards different data structures without needing to entirely re-engineer the data's original forms. For instance, tabular datasets (structured data) and image arrays (unstructured data) can both be simultaneously passed through layers of the same Neural Network as separate inputs, permitting the modeling of complex relationships between structured, semi-structured, and unstructured data. Mixed data in the form of image arrays and CSV files can enter the Network as separate inputs, pass through separate layers for feature extraction during training, and eventually merge into one combined vector to output some type(s) of result(s).

The Role of GPUs and Cloud Computing in Accelerating Neural Network Training

Neural Networks often operate over multidimensional arrays, commonly referred to as ***tensors***, for processing large amounts of data. Common array structures include,

- **Grayscale Images**: 2D-matrices denoted by their width and height as a single-color channel,
- **RGB Images**: 3D-arrays that incorporate 3-color matrices for representing the color spectrum,
- **Videos**: 4D-arrays that stack images (frames) from a video in sequential order (with the 4^{th}-dimension indexed by time).

How are Networks optimized for training millions of weight parameters on these massive array-based structures?

The answer lies in GPUs. GPUs are capable of handling complex tasks in parallel with enhanced performance due to their fast graphics-rendering capabilities. Neural Networks are a series of matrix multiplications, transformations, additions, and other algebraic manipulations that are computed as graphs to permit high-speed performance and parallel computing.

Graphs are a series instructions describing the order of algebraic operations on tensors. Neural Networks are graph-based, meaning that their layers, computations, and outputs can be visualized using a simple flow chart. Regardless of the size of the

data and whether it fits into memory, graphs ensure consistent, fast, parallel, and reliable execution of data passing through the Network.

TensorFlow v2.0 and the Keras API

TensorFlow is a massive machine learning and deep learning library developed by Google and released as open-source software that can be integrated with Python, R, JavaScript, and similar languages to develop, evaluate, and deploy state-of-the-art algorithms.

Following the release of TensorFlow v2.0, all updated versions of TensorFlow come packaged with the Keras API, giving programmers an intuitive, high-level interface for developing simple and complex Neural Networks. In TensorFlow, programming Neural Networks required knowledge of its native Python classes, objects, and methods for customizing and developing layers. Following its integration with the Keras API, however, developing Neural Network layers were simplified to one-line method calls complete with customizable hyperparameters. These hyperparameters include adjustments to the number of nodes (neurons), pre-built and custom-made activation functions, and weight initialization options for any given layer in the Network.

Keras features two main APIs, described below:

- **Sequential API**: Simplest Keras API and easy-to-implement; good for simple Networks—unable to construct advanced Neural Networks.
- **Functional API**: Supports development of more advanced Network architectures (i.e., Multi-Input & Multi-Output Neural Networks)

The Python code in (2.1) shows how to create a very simple, densely connected Neural Network using the Sequential and Functional APIs. This example Network accepts a tabular dataset possessing 5-features, passes it through one hidden layer, and outputs the "probability" of that instance belonging to some class for a binary classification problem.

3. Introducing and Preprocessing the SASHELP.CARS Dataset

Python Libraries, Modules, and Packages

Before proceeding further, (3.1) provides a listing of all dependencies required to import, analyze, and process tabular datasets and image arrays in developing Neural Networks.

Python Code (3.1) | Importing Required Python Libraries, Packages, and Modules

```
# TensorFlow Library and its Modules
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam

# DataFrame manipulation library
import pandas as pd

# Scikit-Learn Data preprocessing library
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# Array processing library
import numpy as np

# Visualization library
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

Importing the modules from TensorFlow as listed in (3.1) significantly reduces the amount of redundant function calls stemming from each module's full access path. When constructing a *Dense* layer, for instance, the programmer doesn't need to type the complete *tf.keras.layers.Dense()* to access this specific Network layer, and can instead use *Dense()*.

Pandas is a popular data science library used for efficiently processing and analyzing moderate-sized data. It is used for importing and preparing the CSV dataset for the Network. Scikit-Learn is a machine learning focused library that integrates well with Pandas, making their predefined functions efficient ways of preprocessing data.

SASHELP.CARS Tabular Data

The SASHELP.CARS dataset is a popular data set bundled with and provided by SAS software. It contains information about hundreds of different car models, describing attributes including the vehicle's make, model, type, origin, invoice price, MSRP, horsepower, number of cylinders, engine size, miles per gallon, and additional specifications. This diversity in vehicle attributes makes it a perfect candidate for regression modeling and classification using certain vehicle features.

The code in (3.2) imports the comma-separated value file, inferring the header titles, and printing out the first-5 observations from the dataset.

Python Code (3.2) | Importing the SASHELP.CARS Dataset

```
# Import the CSV file from local pathway and print its first-5 observations:
file_path = str(r"C:\Users\rpala\Downloads\cars.csv").replace("\\", "/")
cars_df = pd.read_csv(file_path, sep=",", header="infer")
cars_df.head(5)
```

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	W
0	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6.0	265	17	23	
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4.0	200	24	31	
2	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4.0	200	22	29	
3	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6.0	270	20	28	
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6.0	225	18	24	

Output Code (3.2)

The dataset contains 428-observations and 15-features of mixed data types. Some features are continuous numeric such as a vehicle's miles per gallon, MSRP, and invoice price, while other features are discrete numeric like the vehicle's number of cylinders. Categorical attributes exist in the form of a vehicle's type, make, model, origin, and drive train.

Only the dataset's numeric attributes are used for the Neural Networks developed in this Paper. However, it is possible to work with the categorical features (in addition to the numeric attributes) through further processing using ordinal encoding and one-hot encoding.

It's often helpful to examine the meta-data for any given dataset, yielding insights into the number of features, their names, variable types, and the number of missing values per column. The meta-data for the SASHELP.CARS dataset is provided in the output of (3.3).

Python Code (3.3) | Meta-Data for the SASHELP.CARS Dataset

```
# Extract column names, variable types, and missing values from dataset:
col_names = cars_df.columns
dtypes = cars_df.dtypes
missing_vals = cars_df.isnull().sum()

# Create a Dictionary object containing information about each column:
meta_data_dict = {"feature_name": col_names,
                  "dtypes": dtypes,
                  "missing_vals": missing_vals}
```

```
# Create a Pandas DataFrame from the Dictionary object, sorted in descending order:
meta_data_df = pd.DataFrame(meta_data_dict).sort_values("missing_vals", ascending=False)
meta_data_df
```

	feature_name	dtypes	missing_vals
Cylinders	Cylinders	float64	2
Make	Make	object	0
Model	Model	object	0
Type	Type	object	0
Origin	Origin	object	0
DriveTrain	DriveTrain	object	0
MSRP	MSRP	object	0
Invoice	Invoice	object	0
EngineSize	EngineSize	float64	0
Horsepower	Horsepower	int64	0
MPG_City	MPG_City	int64	0
MPG_Highway	MPG_Highway	int64	0
Weight	Weight	int64	0
Wheelbase	Wheelbase	int64	0
Length	Length	int64	0

Output Code (3.3)

In Pandas, variables that contain *strings* or *mixed data* are given the *object* type. *Integers* are specified as numeric values without decimals, while *floats* accommodate any numeric value, including integers. There are only 2 missing-values in the dataset, all localized to the Cylinders attribute. Before proceeding, it is necessary to deal with any missing values before passing it through a Neural Network. Either removing the observations or conducting imputation using the feature's average, median, or through decision tree methods all correct for missing values. Failure to address this results in gradient descent producing *NaN* values with respect to the loss function and learning nothing over its training epochs.

The code in (3.4) drops any missing observations and then finds, extracts, and subsets all columns that match as numeric (*float* or *integer*) types. Notice, however, that MSRP and invoice price are wrongly encoded as string *objects*, rather than float types. To correct for these recording issues, regular expressions (regex) are employed for removing "$" and "," in price strings so that vehicles' MSRPs and invoice prices are successfully coerced from string objects to float types for modeling.

Python Code (3.4) | *NA*-Removal, String Coercion, and Numeric Sub-setting

---

```
# Drop missing values from the entire DataFrame:
cars_df = cars_df.dropna()

# Remove string formatting and characters, then coerce MSRP & Invoice to Float Types:
cars_df["MSRP_num"] = cars_df["MSRP"].str.replace(r"[$,]", "", regex=True)
cars_df["MSRP_num"] = cars_df["MSRP_num"].astype(float)

cars_df["Invoice_num"] = cars_df["Invoice"].str.replace(r"[$,]", "", regex=True)
cars_df["Invoice_num"] = cars_df["Invoice_num"].astype(float)

# Find all relevant Numeric Attributes and extract their column names:
numeric_features = cars_df.select_dtypes(include=['int', 'float']).columns.tolist()

# Subset existing DataFrame to only include Numeric Attributes:
cars_numeric = cars_df[numeric_features]
```

```
# Print the first 5-observations from the numeric features DataFrame:
cars_numeric.head(5)
```

	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	Weight	Wheelbase	Length	MSRP_num	Invoice_num
0	3.5	6.0	265	17	23	4451	106	189	36945.0	33337.0
1	2.0	4.0	200	24	31	2778	101	172	23820.0	21761.0
2	2.4	4.0	200	22	29	3230	105	183	26990.0	24647.0
3	3.2	6.0	270	20	28	3575	108	186	33195.0	30299.0
4	3.5	6.0	225	18	24	3880	115	197	43755.0	39014.0

Output Code (3.4)

4. Artificial Neural Network (ANN) for Non-Parametric Regression

Neural Network Regression Design, Applications, and Tasks

Artificial Neural Networks for regression can accept large quantities of input features (including mixed data types), pass them through several densely connected hidden layers, and use those connections to predict some numeric estimate for the response variable.

Densely connected layers are exhaustive connections between layers. For example, suppose that one layer contains 3 neurons, and the following layer contains 4 neurons, then the total number of edges formed is 12, since each neuron in the preceding layer connects to every neuron in the following layer. The equation shown below derives the total number of trainable weights between the two proposed layers:

$$(3)(4) = 12\ connections\ (weights)$$

The total number of trainable parameters shared between the two layers, which includes their *weights* and associated *biases,* is calculated as,

$$(3+1)(4) = 16\ trainable\ parameters$$

Bias, like the weights, are parameters that self-adjust as the Network trains. A biased estimator is defined as one that favors some incorrect value as opposed to an unbiased estimator which, on average, generally favors the correct result. Every weight has a bias term that is associated with it, and the weight's bias is a value that prevents the training data from exerting too much influence on the Network's layered activations and counteracting the potential for severe overfitting.

Another term for *densely connected* is *fully connected*, since neurons are completely connected to each other in the next immediate layer of the Network. These types of Neural Networks feature trainable parameters that can increase dramatically, resulting in large quantities of weights, and thereby increasing the time it takes to calculate the gradient (using backpropagation) and perform gradient descent for weight optimization.

Using the SASHELP.CARS dataset, an ANN was trained to predict a vehicle's MSRP (response variable) using several numeric features including its engine size, number of cylinders, total horsepower, city MPG, highway MPG, invoice price, total weight, wheelbase, and length (encompassing the set of predictors).

Shuffling, Partitioning, & Preparing Data for Neural Networks

Evaluating Network training should always be done using a validation set—a subset of the testing data that is passed through the Network *only* at the end of each epoch.

As mentioned earlier, gradient descent works through batches of training data. Batch sizes are pre-determined by the programmer, with mini-batch gradient descent mitigating the variability of constant weight updates (unlike stochastic gradient descent that updates weights after 1-training instance passes through the Network). An epoch is completed once all the batches pass through the Network and update its weights. After this, the validation dataset is then evaluated against the loss function using the updated weights from the last batch of data that completed the epoch.

The code in (4.1) splits the existing dataset into a training set, validation set, and testing set. The entire dataset of numeric attributes is randomly shuffled, and the target variable (response variable) for MSRP is then removed from the dataset as its own vector. The training-testing split is 65%: where 35% of the shuffled data is reserved for the testing set.

Furthermore, the testing set is then split such that 80% creates the validation set and the remaining 20% is kept isolated from the model.

Therefore, the randomly shuffled and partitioned data maintains the following proportions: 65% reserved for Network training, 28% for Network validation, and 7% held out for testing after training.

The numeric attributes are then standardized using a method (implemented in Scikit-Learn) called *Minimum-Maximum Standardization*. Standardizing different features that are measured on various scales significantly optimizes gradient descent performance!

Python Code (4.1) | Shuffling, Partitioning, and Standardizing the Data

```
# Keep set of predictors together; remove response and store in its own vector:
X = cars_numeric.loc[:, cars_numeric.columns != "MSRP_num"]
y = cars_numeric["MSRP_num"]

# Partition into Training Set and Testing Set using Scikit-Learn:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.35,
        random_state=777)

# Partition into Validation Set and Testing Set using Scikit-Learn:
X_val, X_test, y_val, y_test = train_test_split(X_test, y_test, test_size=0.2,
        random_state=77)

# Define a Method for Min-Max Standardization on all sets of data:
def standardize(X_train, X_val, X_test):
    standardized = MinMaxScaler()
    X_train = np.array(standardized.fit_transform(X_train))
    X_val = np.array(standardized.transform(X_val))
    X_test = np.array(standardized.transform(X_test))
    return X_train, X_val, X_test

# Standardize the sets of arrays:
X_train, X_val, X_test = standardize(X_train, X_val, X_test)
print(f"Overall Data Size: {cars_numeric.shape[0]} observations,\n")
print(f"Training Set Size: {X_train.shape[0]} observations,\n")
print(f"Validation Set Size: {X_val.shape[0]} observations,\n")
print(f"Testing Set Size: {X_test.shape[0]} observations\n")
```

```
Overall Data Size: 426 observations,

Training Set Size: 276 observations,

Validation Set Size: 120 observations,

Testing Set Size: 30 observations
```

Output Code (4.1)

Developing the ANN's Architecture

The Artificial Neural Network developed for predicting some vehicle's MSRP contains:

- One input layer accepting observations described by 9-features (the number of features in the predictor set),
- 6-densely connected hidden layers with arbitrary numbers of neurons,
- One output layer containing a single neuron for regression predictions.

The deeper a Network is, the better it is at recognizing finer-detailed trends, patterns, and details mapping the set of predictors to the response variable. The more neurons a layer contains, the greater its number of trainable parameters are—higher quantities of weight parameters build a more complex model. Like a linear regression, introducing more information (i.e., additional features and combinations of features) into the Network increases its fit to the response variable.

The code in (4.2) generates an Artificial Neural Network architecture using the Keras Functional API. This API gives the programmer control as to which layers accept what as inputs. Following a call to each layer, the programmer must then specify which layer(s) feed into the current layer. For this example, the Network is a 1-directional graph (*feed-forward flow*) where the preceding layer's output is the following layer's input.

Using the SASHELP.CARS dataset, the ANN was trained to predict a vehicle's MSRP (response variable) using several numeric features including its engine size, number of cylinders, total horsepower, city MPG, highway MPG, invoice price, total weight, wheelbase, and length (encompassing its set of predictors).

Python Code (4.2) | Designing the ANN Architecture

```
# Specify Input Layer Shape:
input_layer = Input(shape=(X_train.shape[1],), name="Input")

# Pass the Input Layer into the First Dense Layer:
dense_1 = Dense(9, activation="relu", name="Dense_1") (input_layer)

# Pass the First Dense Layer into the Next Dense Layer:
dense_2 = Dense(20, activation="relu", name="Dense_2") (dense_1)

dense_3 = Dense(32, activation="relu", name="Dense_3") (dense_2)

dense_4 = Dense(20, activation="relu", name="Dense_4") (dense_3)

dense_5 = Dense(7, activation="relu", name="Dense_5") (dense_4)

dense_6 = Dense(3, activation="relu", name="Dense_6") (dense_5)

# Output Layer contains only 1-neuron for linear regression prediction:
output_layer = Dense(1, activation="linear", name="Output") (dense_6)

# Create the ANN by linking the Network's Input Layer to its Output Layer:
network_1 = Model(inputs=[input_layer], outputs=[output_layer])

# Print out the ANN layer-flow summary:
network_1.summary()
```

```
=================================================================
Total params: 1797 (7.02 KB)
Trainable params: 1797 (7.02 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Output Code (4.2)

The code in (4.2) generates the architecture for a 6-layer deep Network—a simple ANN capable of accepting tabular data containing 9-features and outputting a single prediction for a vehicle's MSRP value. The Network contains a total 1,797 trainable weights and biases that determine whether a neuron is activated or not.

Each layer also possesses an *activation function*. These functions transform Neural Networks from sequences of linear matrix operations into non-linear, complex relationship mappers. One function, the non-linear *Rectified Linear Unit (ReLU) function*, uses

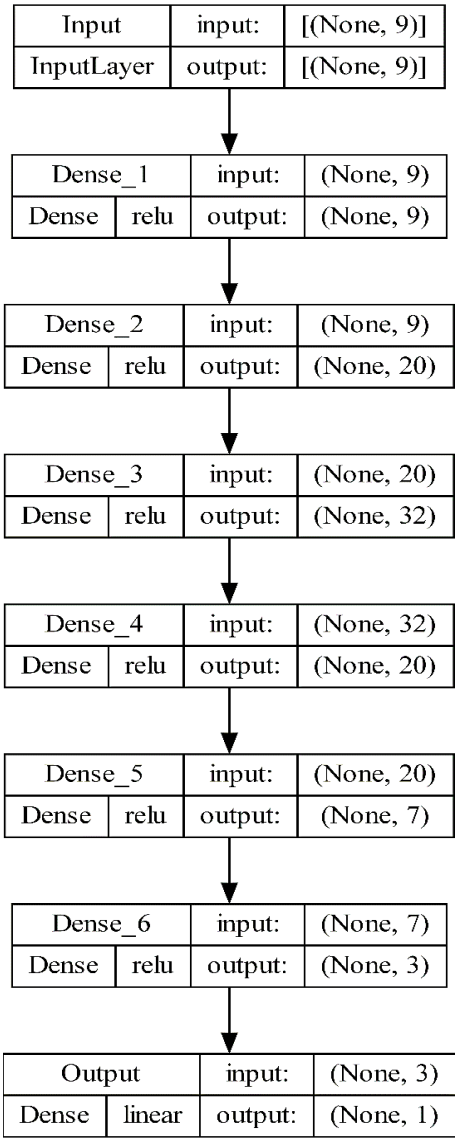

Figure 1. Feedforward Neural Network

the weighted sum of weights multiplied by their inputs as its input. Depending on the weighted sum, the *ReLU* function either outputs 0 for a negative weighted sum or the summation itself if the weighted sum is greater than 0. Choosing the right activation function is critical for gradient descent to work properly. *ReLU* is a popular choice when constructing hidden layers.

Figure 1 shows the forward-passing flow of the Artificial Neural Network constructed in (4.2). The input layer accepts observations containing data about 9-features, which is then passed to the first Dense layer containing 9-neurons used for calculating the output from the layer's *ReLU* non-linear activation function. The original data continues to be transformed and passed through additional hidden layers until a single linear prediction is made in the output layer.

Compiling and Training the ANN

With the architecture fully developed and the layers linked together, programmers can then compile their model by specifying the loss functions, optimizers, and metrics that co-inside with training and evaluation.

The *loss function* is how the Network learns. For this regression task, the Mean Absolute Error (MAE) is a good function for performing backpropagation and mini-batch gradient descent.

The *optimizer* represents the rate at which the Network learns over training epochs. While gradient descent focuses on calculating the loss function's partial first derivative, the optimizer controls the rate at which gradient descent optimizes trainable parameters (similar concept to a function's second derivatives). A key hyperparameter to any optimizer is the Network's adjustable *learning rate*, with small rates leading to longer, more stable training epochs and larger rates leading to shorter but more volatile training epochs.

Metrics are used to evaluate the Network's performance on the training and validation datasets without explicitly training the model on them. Like loss functions and optimizers, programmers can implement pre-made metrics developed by TensorFlow and even make their own metrics using TensorFlow Functions, where Python methods can be optimized for lazy computation using the *@tf.function* decorator.

The code in (4.3) compiles the model using the Mean Absolute Error (MAE) loss function, the Adam optimizer with the learning rate set to 0.001 (its default value), and the Root Mean Squared Error (RMSE) to measure its performance on the training and validation datasets. It also passes the prepared training and validation data to the Network in batches containing 64-instances for 2,000 total epochs. An *early stopping* callback mitigates the issue of overfitting by stopping training once the validation loss doesn't significantly reduce over 30 epochs.

Python Code (4.3) | Compiling and Training the ANN

```
# Network compilation with loss function, optimizer, and metrics:
network_1.compile(optimizer=Adam(learning_rate=0.001),
                  loss=tf.keras.losses.MeanAbsoluteError(),
                  metrics=[tf.keras.metrics.RootMeanSquaredError()])

# Early Stopping Callback:
es = EarlyStopping(monitor='val_loss',
                   mode='min',
                   patience=30,
                   restore_best_weights=True)
```

```
# Train the Network and store information on loss and metrics over each epoch:
history = network_1.fit(X_train, y_train,
                        validation_data = (X_test, y_test),
                        callbacks=[es],
                        epochs=2000,
                        batch_size=64,
                        verbose=1)
```

Evaluating the ANN's Training Performance

One method for evaluating a Network's performance is plotting its training loss against its validation loss. These are called the Network's *learning curves* that track the Network's performance over all training epochs. The plot for this Network is shown in Figure 2.

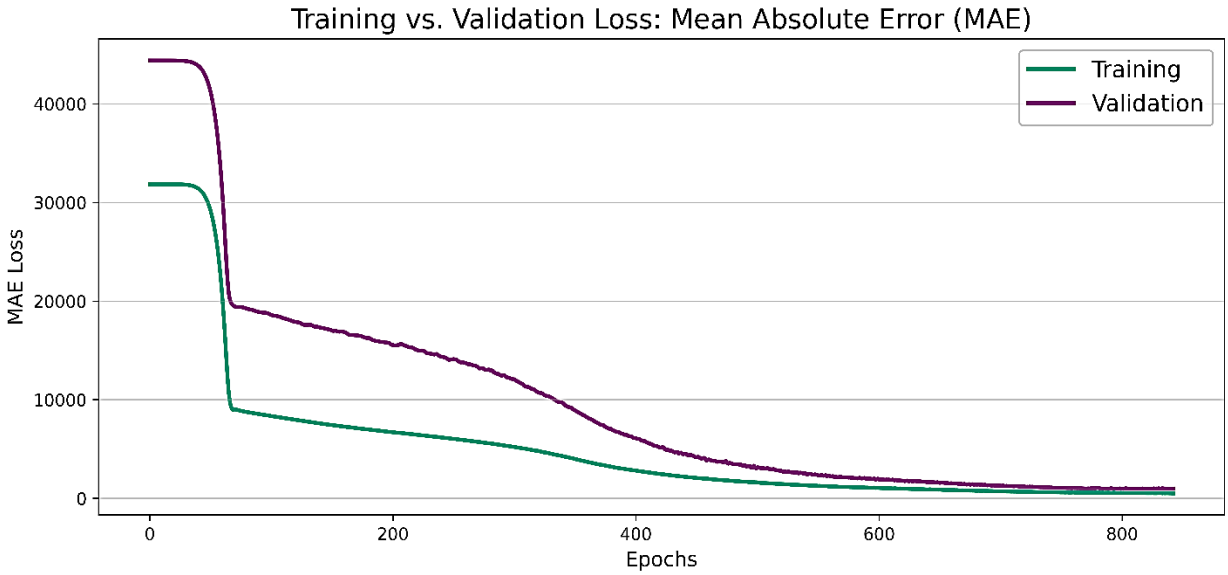

Figure 2. Learning Curves for MAE Loss Function

In Figure 2, the training curve is shaded green, and the validation curve is shaded purple. As the number of training epochs increases, both curves decrease with respect to the Mean Absolute Error loss function, suggesting that the Network's predictive power improves over training.

Also notice how the difference between the validation and training curves is at first large. The training curve, representative of the average absolute loss (*Mean Absolute Error*) in the training data, is consistently lower than its validation counterpart. This is typical for Neural Networks! Since the Network repeatedly learns from the same training data, its loss function will *usually* be lower than the data used to evaluate the Network at the end of each epoch.

If the validation curve steadily converges towards the training curve over epochs, then it suggests the Network is training properly and generalizing well to unseen data. The Network's complexity, defined by its number of neurons, layers, and trainable parameters, is appropriately defined given its architecture.

If the validation curve fails to converge to the training curve, then the Network is at-risk of *overfitting* the training data and failing to generalize. While it performs spectacularly well on the training set, its predictive power on unseen data will fail miserably—instead of learning the mappings between its set of predictors and target feature(s), the Network could instead be learning *noise* inherent to the training data, and only the training data.

Another scenario is if the training curve fails to significantly decrease over epochs. This might suggest that the Network is *underfitting* the data and is failing to capture complex relationships between the set of predictors and response variable. Increasing the number of layers and neurons, changing the activation functions, and augmenting the data to create additional training instances may remedy this issue.

The code for generating the set of learning curves shown in Figure 2 is provided in (4.4).

Python Code (4.4) | Generating Learning Curves from the Network's Training History

```
# Store the Network's training history in a Python Dictionary:
history_dict = history.history

# Extract training loss and validation loss, calculate number of epochs:
train_loss = history_dict["loss"]
val_loss = history_dict["val_loss"]
epochs = range(0, len(train_loss), 1)

# Plot the learning curves using MatPlotLib:
fig, ax = plt.subplots(figsize=(12, 5))
ax.plot(epochs, train_loss, c="#009E60", lw=2.3, zorder=0, label="Training")
ax.plot(epochs, val_loss, c="#702963", lw=2.3, zorder=1, label="Validation")
ax.set_title("Training vs. Validation Loss: Mean Absolute Error (MAE)", fontsize=16)
ax.set_ylabel("MAE Loss", fontsize=12)
ax.set_xlabel("Epochs", fontsize=12)
ax.legend(fontsize=13.5)
ax.yaxis.grid(True, linewidth=0.77, alpha=0.42)
fig.savefig("reg_learning_curves.png", dpi=1000)
plt.show()
```

5. Convolutional Neural Network (CNN) for Image Classification

Overview of Image Classification

Convolutional Neural Networks (CNNs) perform classification and regression tasks in similar fashion to densely connected Artificial Neural Networks with one major exception: inputs do not need to be flattened to vectors prior to passing through hidden layers. Instead, 2D-matrices, 3D-arrays, and multidimensional arrays (i.e., sequences of images from videos, arrays of spatiotemporal data) can pass through the first input layer with their original dimensions kept intact.

This unique property is quite different from traditional Machine Learning algorithms like random forests (RFs), gradient boosted models (GBMs), and support vector machines (SVMs), all of which require data flattening prior to model training. This property also allows Networks to learn patterns, extract fine-detailed features, and recognize objects within images over epochs of training. Classification, regression, object detection, object segmentation, dimensionality reduction, forecasting, and generative applications are all possible with convolutional layers, making them powerful components to many Neural Networks.

The Keras API supports 1D, 2D, and 3D convolutional layers for learning patterns and relationships in vectors, matrices, and 3D-arrays.

Kernels, Convolutions, and Layers: *Detecting Patterns in Images*

The main component to any convolutional layer is its *kernel*. The kernel is a small vector, matrix, or array that moves across elements inside the input array, capturing a subset of elements within the kernel's window, and calculating a weighted sum that reduces the overall size of input array while increasing its depth.

An array's depth is how deep its dimensions are. For any typical color image, the depth of its array is 3-channels deep, representing the 3-matrices mixing red, green, and blue to produce any color on the visible color spectrum. A grayscale image, on the other hand, is a 2D-matrix that only possesses a depth of 1-channel.

The output of any convolutional layer is an array whose shape possesses a greater depth from the kernel scanning its pixels, extracting important features, and resizing it.

For example, suppose that a high definition RGB-image of size (1920*x*1080*x*3) is passed through a convolutional layer containing a kernel of size (2*x*2) that results in 10-new features (called **filters** in TensorFlow) being created. Following this convolution, the resulting array is resized to dimensions (1919*x*1079*x*10), where the original 3-color channels are mixed to create 10-distinct features (patterns, objects, etc.) from all areas of the image. The convolution resulted in the array's depth increasing from 3-channels to 10-channels, yielding new information about the image's features mapped by this layer.

Deep Convolutional Neural Networks contain stacks of convolutional layers where the original input is passed through a series of layers, transformed during feature extraction, and output as some result.

Classifying Images as Either "Sandstorms" or "Fog/Smog" Scenes

The images used in developing the binary image classification CNN for this paper contained scenes of natural weather phenomena from around the world. These images consisted of labeled scenes showing either snow, rain, frost, lightning, sandstorms, and fog/smog events. For simplicity, the images were filtered to only those showing sandstorm and fog/smog events for the purposes of binary classification—although it's entirely possible to develop a Neural Network that predicts the probabilities of an image belonging to more than two distinct classes. These types of Networks can solve multi-classification problems.

The images themselves were of different sizes and resolutions—necessitating preprocessing to ensure all images were resized (scaled) to the same dimensions and converted from 3D-arrays to 2D-grayscale scenes.

The best way of facilitating large images entering Python for training, without occupying unnecessary space residing in memory, was to construct a TensorFlow pipeline that extracted all images from their labeled file locations, lazily-loaded them as chunks into memory, and performed preprocessing with parallel computing (where each CPU was designated as a *worker* completing tasks in parallel with other *workers*).

Lazy evaluation permits larger-than-memory datasets to be loaded into Python without it crashing from lack of RAM (Random Access Memory). Instead of loading all the image arrays into Python at once, lazy loading facilitates only a small subset of images (chunks) into memory at-any-time. Image chunks are then efficiently processed as TensorFlow Datasets.

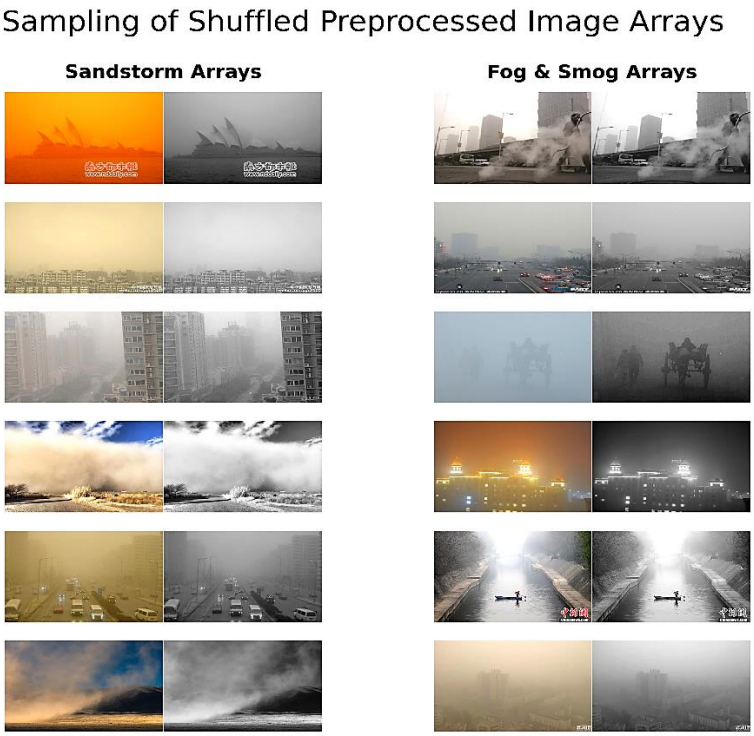

Figure 3. Random Sample of Processed Sandstorm and Fog/Smog Scenes with their Grayscale Versions

This preprocessing pipeline involved first resizing images from their original dimensions to a common size of (50x150) pixels. They were then normalized to contain pixel values between [0, 1] and converted from RGB-arrays to grayscale matrices to speed-up Network training. Figure 3 shows the original images compared to their grayscale counterparts, labeled according to their scene, and then randomly shuffled before being split into their training, validation, and testing datasets.

Developing a Typical CNN Architecture

Processing these grayscale images for binary classification as *sandstorm* or *fog/smog* events requires several hidden layers that convolve, pool, flatten, and eventually pass those flattened inputs through dense layers to obtain classification probabilities.

There are several methods for pooling arrays—this involves *aggregating* nearby pixels to reduce the size of arrays by calculating that pixel group's average, median, minimum, or maximum value and replacing those pixel values with the aggregated value. Maximum (*max*) pooling is particularly effective because it aggregates small groups of pixels by their biggest, and often, most interesting, pixel value. When used in combination with convolutional layers, pooling *downscales* (decreases the image's resolution) the number of pixels in its input to extract additional features and produce arrays containing informative feature channels. These feature channels are *abstract representations* of the original image, often examining distinct sections of the image to find patterns, prominent features, and additional information to assist with the classification task.

Convolutional and pooling layers are sequentially added to the CNN until the data array is small enough to be vectorized (*flattened*), then passed through layers of densely connected neurons, and eventually connected to a single neuron layer that outputs a single class probability. By continuously downscaling the image array, and detecting patterns at each lower resolution, the CNN learns distinct feature representations before being deconstructed to a single column (vector) of flattened data.

A CNN architecture that accepts images of dimensions (50*x*150) contains 9-hidden layers that convolve, pool, flatten, and densely connect the original input to its desired output—a binary probability score that classifies an image as a "sandstorm" or "fog/smog" scene. The code for developing this CNN with the Keras Functional API is provided in (4.5).

Python Code (4.5) | Developing the CNN Architecture using the Keras Functional API

```
# Specify length, width, and number of channels for image input:
im_length = 50
im_width = 150
channels = 1

# Specify expected input shape of grayscale images:
inputs = Input(shape = (im_length, im_width, channels),
               name = "image")

# Convolutional layer with (2 x 2) kernel → produces 3-features from its input:
conv_1 = Conv2D(filters=3,
                kernel_size=(2, 2),
                padding="valid",
                activation="relu",
                name="conv_1") (inputs)

# Max-Pooling layer that reduces input size by a factor of 2:
pooling_1 = MaxPooling2D(pool_size=(2, 2),
                         padding="valid",
                         name="pooling_1") (conv_1)

# Convolutional layer with (3 x 3) kernel → produces 10-features from its input:
conv_2 = Conv2D(filters=10,
                kernel_size=(3, 3),
                padding="valid",
                activation="relu",
                name="conv_2") (pooling_1)

# Max-Pooling layer that reduces input size by a factor of 2:
pooling_2 = MaxPooling2D(pool_size=(2, 2),
                         padding="valid",
                         name="pooling_2") (conv_2)

# Convolutional layer with (2 x 2) kernel → produces 15-features from its input:
conv_3 = Conv2D(filters=15,
                kernel_size=(2, 2),
                padding="valid",
                activation="relu",
                name="conv_3") (pooling_2)

# Max-Pooling layer that reduces input size by a factor of 2:
pooling_3 = MaxPooling2D(pool_size=(2, 2),
                         padding="valid",
                         name="pooling_3") (conv_3)
```

```
# Flattens array input to a vector → passed as input to Dense layers:
flatten = Flatten(name="flatten") (pooling_3)

# Densely connected layer with 80-neurons; ReLU activation:
dense_1 = Dense(80, activation="relu") (flatten)

# Densely connected layer with 40-neurons; ReLU activation:
dense_2 = Dense(40, activation="relu") (dense_1)

# Output layer with sigmoid activation → produces a single class “probability”:
outputs = Dense(1, activation="sigmoid") (dense_2)

# Stitches the CNN together from input to output layers:
network = Model(inputs=[inputs], outputs=[outputs])
network.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image (InputLayer)          [(None, 50, 150, 1)]      0

 conv_1 (Conv2D)             (None, 49, 149, 3)        15

 pooling_1 (MaxPooling2D)    (None, 24, 74, 3)         0

 conv_2 (Conv2D)             (None, 22, 72, 10)        280

 pooling_2 (MaxPooling2D)    (None, 11, 36, 10)        0

 conv_3 (Conv2D)             (None, 10, 35, 15)        615

 pooling_3 (MaxPooling2D)    (None, 5, 17, 15)         0

 flatten (Flatten)           (None, 1275)              0

 dense (Dense)               (None, 80)                102080

 dense_1 (Dense)             (None, 40)                3240

 dense_2 (Dense)             (None, 1)                 41

=================================================================
Total params: 106271 (415.12 KB)
Trainable params: 106271 (415.12 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Output from Code (4.5)

Examining the Output from (4.5), notice how the pooling layers reduce the input's original size by (approximately) a factor of 2. It works alongside the convolutional layers to assist in feature extraction at different resolutions of the input array. The further the array travels through the CNN, the smaller its original dimensions become while its number of channels significantly increases.

The largest number of trainable parameters occurs after flattening, where the array of size (5*x*17*x*15) is vectorized and passed through a densely connected layer containing 80-neurons. The final output layer, which contains only 1-neuron, possesses a different activation function than the previously defined Artificial Neural Network for regression. The activation function used for binary classification is the *sigmoid function*—the same function used in logistic regression! A multi-classification Network (target features containing more than 2-classes) would use a SoftMax activation function for its output layer.

Compiling the CNN for Training

The code in (4.6) compiles the Convolutional Neural Network using the binary cross-entropy loss function, the Adam optimizer, and metrics including binary accuracy, precision, and recall. When fitting the Network to the training data, mini-batches containing 128 randomly shuffled images are passed through the Network's layers prior to the weights being updated.

An early stopping callback was programmed to halt Network training if the validation loss failed to significantly decrease before reaching 500-total training epochs.

Python Code (4.6) | Compiling the CNN and Fitting it to the Training Images

```
# Early Stopping Callback that monitors validation loss:
es = EarlyStopping(monitor='val_loss',
                   mode='min',
                   patience=40,
                   restore_best_weights=True)

# Compile the Network for Binary Classification with metrics:
network.compile(optimizer=Adam(learning_rate=0.001),
                loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                metrics=[tf.keras.metrics.BinaryAccuracy(),
                         tf.keras.metrics.Recall(),
                         tf.keras.metrics.Precision()])

# Record Network training history for 500-epochs, in batches of 128-images, with
# early stopping
history = network.fit(train_data,
                      epochs=500,
                      batch_size=128,
                      validation_data=val_data,
                      callbacks=[es])
```

Evaluating the CNN's Training Performance

The Convolutional Neural Network developed in this Paper was set to train for a maximum of 500-epochs if it did not automatically stop when the validation dataset's loss function failed to significantly decrease. Early stopping did, however, prevent the CNN from overfitting to the training data, halting training at the 376th epoch.

The set of learning curves measuring the binary cross-entropy losses over epochs for the training data and validation data are plotted in Figure 4.

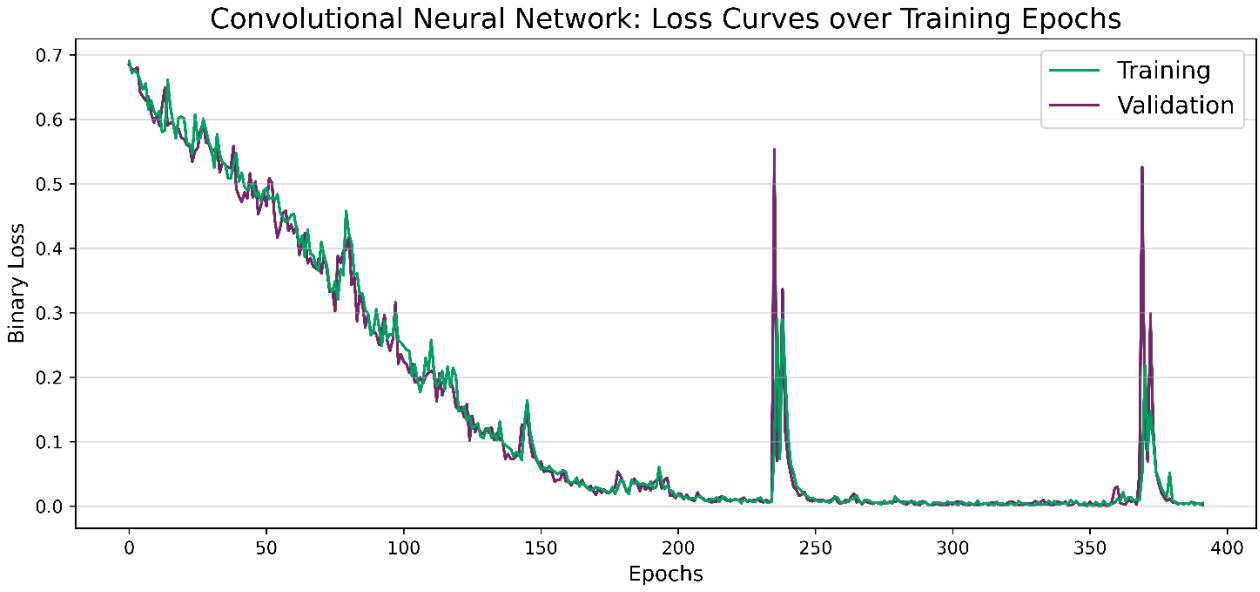

Figure 4. Binary Entropy Training and Validation Loss over Epochs

Examining Figure 4, the training and validation curves closely track to each other, with both loss curves steadily decreasing over initial training epochs. The learning rate was set to its default parameter of *0.001*, leading to steady training over time. Notice, however, that both the training and validation loss curves abruptly spike in later epochs—this is not an error, but rather a feature

of mini-batch (and its more erratic counterpart, stochastic) gradient descent. Depending on how the training and validation instances were randomly shuffled into batches during each training iteration, it is statistically probable for combinations of outlying instances to be grouped into one mini-batch. These *stochastic shocks* can temporarily disrupt learning until the Network corrects itself (by re-optimizing its weights) following additional training epochs containing more "normal" training instances.

Overall, the learning curves show the CNN generalizing well to unseen data in the validation sets, while also learning patterns and features present in the training data. An exemplary CNN!

6. Conclusion

This Paper provides an in-depth discussion on Neural Networks, showcasing their architectures, training processes, and applications to structured and unstructured data through examples programmed in Python with TensorFlow and the Keras API.

Importance of Network Architectures in Deep Learning

Understanding Neural Network architectures are crucial for grasping the intricate mechanisms underlying modern deep learning technology. These composite functions, comprised of interconnected layers of neurons with trainable weight parameters, utilize backpropagation to optimize weights—an automatic differentiation process that is vital for minimizing loss functions. The ability of Neural Networks in handling various types of data with a single model have revolutionized data modeling, eliminating the need for extensive feature engineering and data preparation.

The use of modern-day GPU and cloud computing technologies accelerate Network training, owing to their superiority in parallel processing. Furthermore, user-friendly APIs like Keras that interface with TensorFlow democratize Neural Network development, simplifying class-based programming to methods called with only a few lines of code.

Encouragement for Further Exploration and Experimentation

Neural Networks exhibit remarkable versatility in their modeling capabilities, with two methods presented in this Paper—non-parametric regression and binary image classification. While this Paper primarily focuses on two types of Neural Network architectures, Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs), there are countless types of Neural Network architectures developed for handling different tasks and data structures. Further examples include Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), and Variational Autoencoders (VAEs).

This Paper serves as an example-oriented guide for users seeking a comprehensive understanding of fundamental Neural Network architectures and their practical applications. As the fields of artificial intelligence, machine learning, and deep learning continue to evolve, the insights presented in this Paper give users an understanding of the inner workings developing Neural Networks.

7. About the Authors

Ryan Paul Lafler is the Founder, CEO, Chief Data Scientist, and Lead Consultant at *Premier Analytics Consulting, LLC*, a consulting firm based in San Diego, California, that specializes in optimizing Machine Learning algorithms for Artificial Intelligence workflows; developing responsive Full-Stack Applications and Dashboards; leveraging Open-Source Software for powerful analysis; and offering personalized training tailored to his Clients' Big Data goals. He's also an Adjunct Professor at San Diego State University for the Big Data Analytics Graduate Program and the Department of Mathematics and Statistics. Ryan's multilingual experience in Python, R, SAS, JavaScript (React.js & Node.js), and SQL has contributed to his success as a Big Data Scientist; Machine Learning Engineer; Statistician; Full-Stack Application Developer; and Project Manager. He received his Master of Science in Big Data Analytics from San Diego State University in May 2023 following the successful defense and publication of his thesis. He holds a Bachelor of Science in Statistics and minored in Quantitative Economics from San Diego State University after graduating *Magna cum Laude*. His passions include Machine Learning, Deep Learning, Artificial Intelligence, Statistics, full-stack application and interactive dashboard development, data visualization, and Open-Source programming languages.

Anna Wade is an accomplished statistician, currently working in clinical trials at Medicinova, La Jolla. Anna is also a contractor with *Premier Analytics Consulting, LLC*, as a biostatistician and data scientist. Anna began working in math education shortly following her graduation from the University of California, Santa Barbara in 2019, completing dual bachelor's degrees in mathematics and philosophy. In 2021, while pursuing her Master of Science in Statistics from San Diego State University, she worked as an Instructor and Graduate Teaching Associate for the Department of Mathematics and Statistics, where she found joy in simplifying complex statistical concepts for students, and positively impacting their education experience. Through her studies, she became proficient in SAS, R, and Python, and acquired a profound understanding of mathematical and theoretical statistics.

Since graduating, Anna discovered that her passions lie in many fields including environmental research, marine biology, and climatology. She is dedicated to inspiring the next generation of researchers and scientists, all while advocating for ethical practices, equal opportunities, and environmental stewardship.

Comments, suggestions, and/or any questions may be sent to:

Ryan Paul Lafler, M.Sc.
Premier Analytics Consulting, LLC
CEO, Chief Data Scientist, Lead Consultant, and Adjunct Professor
E-mail: rplafler@premier-analytics.com
Website: https://www.Premier-Analytics.com
LinkedIn: https://www.LinkedIn.com/in/RyanPaulLafler/

Anna Wade, M.Sc.
Premier Analytics Consulting, LLC
Contractor, Biostatistician, and Data Scientist
E-mail: atwade@premier-analytics.com
Website: https://www.Premier-Analytics.com