Feed Forward Neural Network for Intent Classification: A Procedural Analysis

Brady Steele[1]
Brady.Steele@selectquote.com, Bsteele45@gatech.edu

Abstract—This research paper presents an in-depth exploration of a neural network architecture tailored for intent classification using sentence embeddings. The model comprises a feedforward neural network with two hidden layers, *ReLU*[2] activation functions, and softmax activation in the output layer. This paper meticulously examines the technical intricacies involved in data preprocessing, model architecture definition, training methodologies, and evaluation criteria. Detailed explanations are provided for the rationale behind architectural decisions, including the incorporation of dropout[3] layers for regularization and class weight balancing techniques for handling imbalanced datasets. Moreover, the mathematical foundations of the chosen loss function *(sparse categorical cross-entropy)*[4] and optimization algorithm *(Adam optimizer)*[5] are thoroughly elucidated, shedding light on their roles in facilitating model training and convergence. Through empirical experiments and theoretical analyses, this paper offers insights into the effectiveness and resilience of the proposed neural network architecture for intent classification tasks. It serves as a technical guide for engineers aiming to comprehend, implement, and optimize neural network models for practical application in natural language processing endeavors.

1 The author would like to thank SelectQuote and Georgia Institute of Technology.
2 ReLU: Rectified Linear Unit.
3 Dropout: A regularization technique used to prevent overfitting in neural networks by randomly setting a fraction of input units to zero during training.
4 Sparse Categorical Cross-Entropy: A loss function used for multi-class classification tasks with integer labels.
5 Adam Optimizer: An optimization algorithm that combines the benefits of AdaGrad and RMSProp optimizers.

1 INTRODUCTION

1.1 Background and Motivation

A customer service phone platform facilitates interactions by providing automated prompts to customers calling in, followed by their responses. The goal of the neural network discussed in this paper is to accurately classify these prompts and responses into one of the predefined intents, namely: *answered_question*, *off_topic*, *operator_human*, *please_wait*, *repeat_question*, or *more_info*.

1.2 Problem Statement

Intent classification in this domain poses several challenges, including the need to accurately capture the nuanced semantic relationships between the prompts, responses, and intent labels while effectively managing issues such as data sparsity and class imbalance. To address these challenges, the proposed architecture leverages a feedforward neural network paradigm, which facilitates efficient computation and parameter optimization.

The core of the model lies in its mathematical underpinnings, rooted in the principles of linear algebra, calculus, and optimization theory. Through rigorous mathematical formulations, I unravel the intricate processes of forward and backward propagation[6], which underlies the training of the neural network.

1.3 Methodology

Intent classification using neural network architectures involves a systematic process encompassing data preprocessing, model architecture design, training procedures, and evaluation metrics. The below sections provide a detailed overview of the methodology employed in this solution.

2 DATA PREPROCESSING

Data preprocessing is a critical step in preparing the raw input data for training and inference. It involves several sub-tasks, including handling missing values, text tokenization, label encoding, chunking, and generating embeddings.

6 Forward and backward propagation are key processes in training neural networks, involving the calculation of output and error gradients, respectively.

2.1 Handling Missing Values

Missing values within the dataset can introduce noise and hinder the learning process of machine learning models. The following function is designed to address this issue by meticulously parsing the JSON-formatted strings within the prompt column of the comma-separated dataset. Through JSON parsing techniques, relevant information pertaining to agent prompts and user responses is extracted. Furthermore, missing values are identified and managed within the parsing function to ensure the integrity of the dataset. By systematically handling missing values, the preprocessing pipeline mitigates the risk of biased or inaccurate model training.

2.2 Text Tokenization

Within the *Prompt and Response Extraction Function*, JSON-formatted strings sourced from the *prompt* column of the DataFrame are transformed into Python list structures for iterative analysis. Leveraging regular expressions, the function iterates over each prompt component, scrutinizing role and content attributes to discern agent prompts and user responses.

By employing regular expressions via the **'re.search()'** function, specific patterns indicative of agent prompts and user responses are identified wihin the content of each prompt component. Upon pattern identification, the function extracts corresponding text segments, delineating agent prompts and user responses as distinct tokens. This tokenization process lays the groundwork for subsequent feature extraction and model training tasks by segmenting the prompt data into granular units.

Additionally, post-extraction, tokens undergo refinement and standardization to ensure uniformity and consistency in representation, crucial for harmonizing textual data and mitigating potential inconsistencies arising from formatting or context structure variations.

2.3 Label Encoding

The *scikit-learn LabelEncoder*[7] is a versatile tool designed to transform categorical labels into numerical representations, a crucial step in machine learning workflows.

7 The 'LabelEncoder' class in scikit-learn: Documentation.

```
label_encoder = LabelEncoder()
df['encoded_intent'] = label_encoder.fit_transform(df['intent'])
```

Listing 1—Label Encoding

Internally, the *LabelEncoder* initializes an empty dictionary to store the mappings between categorical labels and their corresponding numerical representations. When the *fit()* method is called with the categorical label data, the *LabelEncoder* scans through the unique labels in the dataset and assigns a unique integer to each label, preserving the order in which they appear. This process effectively creates a one-to-one mapping between the categorical labels and the assigned integers with encoded target value labels between *0 and n classes - 1*.

Subsequently, when the *transform()* method is invoked, the *LabelEncoder* utilizes the mappings generated during the fitting stage to convert the categorical labels in new datasets or during predictions into their numerical counterparts. This transformation is achieved by simply looking up each label in the internal dictionary and replacing it with its corresponding integer value.

Labeled Intents	
Intent Name	Intent Label
answered_question	0
off_topic	1
repeat_question	2
please_wait	3
more_info	4
operator_human	5

Table 1—Example implementation of LabelEncoder numerical representations

2.4 Chunking

The chunking process plays a pivotal role in managing memory resources efficiently, particularly pertinent when handling sizable datasets, thereby mitigating the likelihood of memory overflow errors. This technique involves dividing the dataset into smaller, manageable segments, with the size of each segment dictated by the parameter *chunk size*. Mathematically, let's denote:

- N as the total number of samples (or rows) in the dataset.

- C as the chunk size, representing the number of samples in each chunk.
- K as the total number of chunks, calculated as $K = \lceil N/C \rceil$, where $\lceil \cdot \rceil$ denotes the ceiling function, ensuring that any remainder from division is accounted for.

Now, let's define the indexing process for chunking:

- For the kth chunk, the starting index $start_k$ is given by

$$start_k = (k-1) \times C$$

- The ending index end_k for the kth chunk is calculated as

$$end_k = min(N, k \times C)$$

  ensuring that the last chunk may be smaller if N is not perfectly divisible by C.

These indices delineate the boundaries of each chunk within the dataset. The chunking process then iteratively extracts the segments of the dataset based on these indices, ensuring that each segment contains C samples except for the last chunk, which may contain fewer samples if N is not evenly divisible by C.

2.5 Embedding

Subsequently, these extracted chunks are forwarded to an embedding model, denoted as the *embedder*, which orchestrates the conversion of textual inputs into dense numerical representations, commonly referred to as *embeddings*. The embedder, instantiated with the architecture *all-MiniLM-L6-v2*[8], operates as a transformer-based model designed to encode semantic information from the input prompts and responses efficiently.

The *all-MiniLM-L6-v2* architecture embodies a sophisticated combination of transformer layers optimized for semantic encoding of textual data. It comprises several key components:

1. **MiniLM Backbone:**
   - MiniLM represents a variant of the transformer architecture optimized for efficiency in both training and deployment scenarios.

8 The all-MiniLM-L6-v2 SentenceTransformer: Documentation

- The *all-MiniLM-L6-v2* variant specifically denotes a configuration characterized by six transformer layers *(L6)*. These layers are stacked atop one another, forming a deep hierarchical structure for processing input sequences.

2. **Transformer Layers:**
   - Each transformer layer integrates multi-head self-attention mechanisms and position-wise feedforward networks.

   - Multi-head self-attention enables the model to focus on different parts of the input sequence simultaneously, capturing contextual dependencies and semantic relationships across tokens.

   - Position-wise feedforward networks introduce non-linear transformations to the token embeddings, facilitating the extraction of higher-level features and semantic information.

Mathematically, at each layer l , the input embeddings Emb_{l-1} undergo transformations according to the following formulation:

$$Out_l = LayerNorm(MHA(Emb_{l-1}) + Emb_{l-1})$$

where MHA denotes the multi-head self-attention mechanism, and $LayerNorm$ represents layer normalization. This iterative process progresses through each layer until reaching the final transformer layer *(L6)*.

Additionally, the embedder leverages *HuggingFace's SentenceTransformers*[9] framework, which provides a high-level interface for utilizing pre-trained transformer models in tasks related to sentence embeddings. The framework offers a wide range of pre-trained models, including *all-MiniLM-L6-v2*, facilitating seamless integration into various NLP pipelines.

HuggingFace's SentenceTransformers framework abstracts away the complexities of model loading, tokenization, and inference, allowing researches and practitioners to focus on their specific NLP tasks. Behind the scenes, the framework handles the instantiation of pre-trained models, tokenization of input text, and computation of sentence embeddings using efficient GPU-accelerated implementations.

For each chunk of prompts and responses, the embedding model orchestrates

9 HuggingFace SentenceTransformers: Documentation

the generation of embeddings for individual sequences, culminating in sets of prompt and response embeddings. These embeddings, stored within lists designated as *prompt embeddings* and *response embeddings*, respectively, represent condensed numerical representations that encapsulate the semantic nuances of the textual data.

2.6 Principal Component Analysis

Principal Component Analysis (PCA) is a linear transformation technique used to reduce the dimensionality of high-dimensional data while preserving as much variance as possible. Given a dataset X consisting of n observations of d-dimensional vectors $x_1, x_2, ..., x_n$ where $x_i \in R^d$, *PCA* aims to find a set of orthonormal vectors[10] $v_1, v_2, ..., v_d$, known as principal components, that capture the maximum variance in the data.

1. Center the data by subtracting the mean vector $\bar{x}$:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$x_i' = x_i - \bar{x}$$

2. Compute the covariance matrix[11] C of the centered data:

$$C = \frac{1}{n}\sum_{i=1}^{n} x_i' x_i'^T$$

3. Find the eigenvectors $v_1, v_2, ..., v_d$ and eigenvalues[12] $\lambda_1, \lambda_2, ..., \lambda_d$ of C. Sort the eigenvectors in descending order of their corresponding eigenvalues.
4. Project the centered data onto the principal components to obtain the transformed dataset:

$$Y = X'V$$

   - Y is the transformed dataset.
   - X' is the centered data matrix.
   - $V = [v_1, v_2, ..., v_d]$ is the matrix of eigenvectors.

10 Orthonormal vectors: A set of vectors that are orthogonal (perpendicular) to each other and have a length of 1.

11 Covariance matrix: A matrix that summarizes the covariance between multiple variables.

12 Eigenvectors and eigenvalues: Concepts from linear algebra representing special vectors and scalars associated with a square matrix.

The scatterplot of PCA-transformed data visualizes the principal components in a lower-dimensional space. Each point in the scatterplot represents a data point projected onto the principal components. The scatterplot reveals semantic relationships among sentences. Clusters of points indicate groups of semantically similar sentences, while the spatial arrangement of points reflects underlying semantic structures captured by the embedding model.

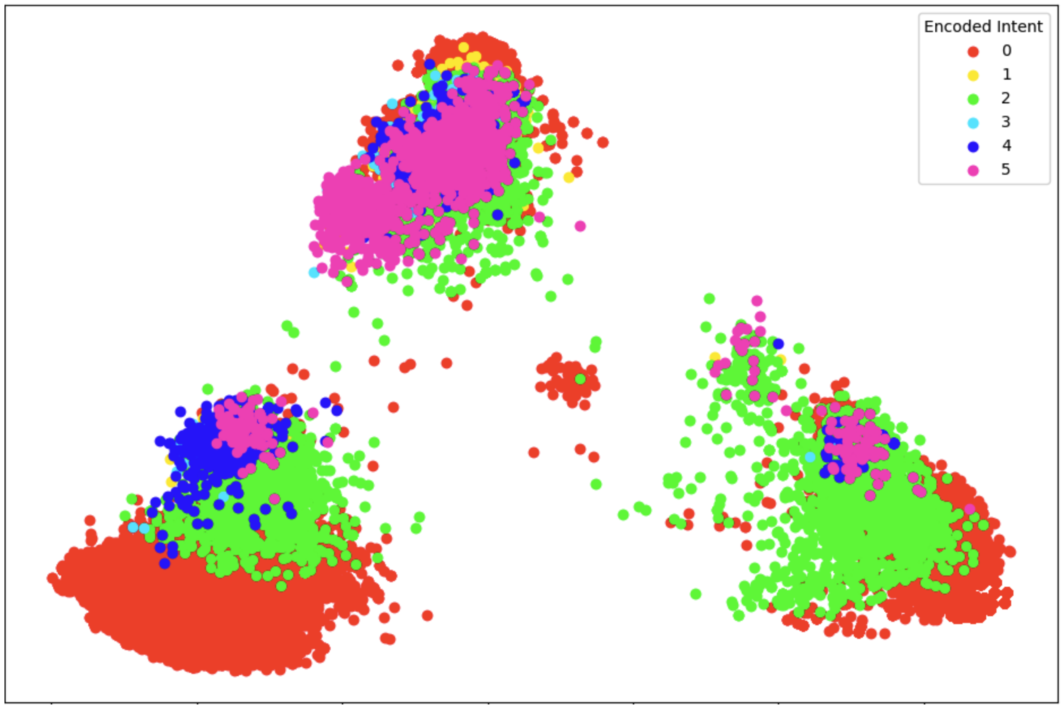

Figure 1—Principal Component Analysis Scatterplot

3 MODEL ARCHITECTURE

In this section, the sophisticated design decisions and formulations that construct the foundation of each layer within the intent classification neural network are examined. From a broad overview of a standard feedforward neural network, to an analysis of the mathematical underpinnings of each component, the inner workings of the architecture are unveiled, demonstrating its adaptability and efficacy in discerning complex patterns within the data.

3.1 Overview of Feedforward Neural Networks

Feedforward neural networks (FNNs), also known as *multilayer perceptrons (MLPs)*, represent a fundamental class of artificial neural networks characterized by their unidirectional flow of information[13]. The architecture of an FNN consists of multiple layers[14] of interconnected neurons[15], each layer comprising a set of nodes responsible for processing and transforming input data. The mathematical breakdown of these networks is as follows:

1. **Layer-wise Computation**
   - Let X denote the input data, a vector of features, and X_{ij} represent the *i*th feature of the *j*th data point.

   - Each neuron in the input layer receives the input features and performs a weighted sum of the inputs along with the bias term b. Mathematically, the output of neuron i in the input layer can be represented as:

$$Z_i^{(1)} = \sum_{j=1}^{n} W_{ij}^{(1)} X_{ij} + b_i^{(1)}$$

   - $W_{ij}^{(1)}$ denotes the weight connecting the *j*th input feature to the *i*th neuron in the first hidden layer.
   - $b_i^{(1)}$ is the bias term associated with neuron i.

2. **Activation Function**
   - After computing the weighted sum, an activation function is applied element-wise to introduce non-linearity into the network's computations. Common activation functions include the *rectified linear unit (ReLU), sigmoid*, and *hyperbolic tangent (tanh)* functions. Below are example implementations for each of the above:
   - **Rectified Linear Unit (ReLU)**

$$f(x) = max(0, x)$$

     Graphically, the ReLU function looks like a linear function for positive inputs and flat lines for negative inputs, as it only passes positive values through unchanged.

13 Unidirectional flow of information: Refers to the flow of data through the network from input to output without any feedback loops.

14 Layers: Structures within neural networks comprising interconnected neurons that process and transform input data.

15 Neurons: Basic units of computation within neural networks, analogous to biological neurons.

- **Sigmoid Function**

$$f(x) = \frac{1}{1 + e^{-x}}$$

  The sigmoid function maps any real-valued number to a range *(0,1)*, making it suitable for binary classification tasks where the output represents the probability of belonging to one class.

- **Hyperbolic Tangent (tanh) Function**

$$f(x) = \frac{e^{x} - e^{(-x)}}{e^{x} + e^{(-x)}}$$

  The tanh function maps any real-valued number to the range *(-1,1)*, making it suitable for classification tasks where the output needs to be symmetric around zero.

3. **Hidden Layers**
   - In a feedforward neural network, one or more hidden layers exist between the input and output layers. Each hidden layer performs a series of computations similar to the input layer, with the output of one layer serving as the input to the next. Mathematically, the output of neuron *i* in the *k*th hidden layer can be expressed as:

$$Z_i^k = f\left(\sum_{j=1}^{m} W_{ij}^k Z_j^{(k-1)} + b_i^k\right)$$

   - $W_{ij}^{(k)}$ denotes the weight connection neuron *j* in the *(k-1)*th layer to neuron *i* in the *k*th layer.
   - $Z_j^{(k-1)}$ is the output of neuron *j* in the *(k-1)*th layer.
   - $b_i^{(k)}$ is the bias term associated with neuron *i* in the *k*th layer.

4. **Output Layer**
   - The output layer aggregates the information from the final hidden layer and produces the network's output. For classification tasks, the output layer typically consists of neurons corresponding to the number of classes in the dataset, with each neuron representing the probability of belonging to a particular class. The *softmax* function is commonly used to compute these probabilities, ensuring that the outputs sum to one.

   - Mathematically, the output of neuron *i* in the output layer can be expressed as:

$$Z_i^{(L)} = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- z_i represents the weighted sum of inputs to neuron *i*.
- *C* is the total number of classes.
- The *softmax* function computes the probability of class *i* given the inputs.

3.1.1 *Softmax Activation Function*

The softmax activation function is a fundamental component of neural network architectures, commonly used in the output layer to generate probability distributions over multiple classes.

Definition

- Given an input vector

  $$z = (z_1, z_2, ..., z_n)$$

  the softmax function transforms the raw logits into a probability distribution over *n* classes. The softmax activation function for the *i*th class is defined as:

  $$P(y_i|z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

  - $P(y_i|z)$ represents the probability of the *i*th class given the input logits *z*.
  - *e* denotes Euler's number (approximately 2.71828).
  - z_i is the raw logit associated with the *i*th class.
  - $\frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$ is the sum of exponential values over all classes.

Properties

1. **Normalization:** The softmax function ensures that the output probabilities sum to one, guaranteeing a valid probability distribution:

   $$\sum_{i=1}^{n} P(y_i|z) = 1$$

   This property is crucial for interpreting the output of the neural network as class probabilities.

2. **Positivity:** Since the softmax function involves exponentiation, the output probabilities are non-negative for all *i*:

$$P\left(y_i|z\right) \geqslant 0$$

3. **Monotonicity:** The softmax function is monotonically increasing. As the input logits increase, the corresponding probabilities also increase:

$$z_i \leqslant z_j \Rightarrow P\left(y_i|z\right) < P\left(y_j|z\right)$$

   This property ensures that higher logit values correspond to higher probabilities, preserving the ordinal relationship between classes.

4. **Interpretation:** The softmax function transforms the raw output of the neural network into a probability distribution, allowing for intuitive interpretation and decision-making. By converting logits into probabilities, the softmax activation enables the model to output confident predictions about the likelihood of each class.

5. **Application:** In practice, the softmax function is used in multi-class classification tasks, where the goal is to assign input data to one of several possible categories, such as in intent classification. The output probabilities produced by softmax facilitate decision-making by identifying the most probable class for a given input.

6. **Limitations:** In cases of highly imbalanced datasets or when dealing with noisy or ambiguous inputs, softmax probabilities may not accurately reflect the underlying uncertainty in the data. Additionally, softmax outputs are sensitive to outliers and large input values, potentially leading to unstable gradients during training.

3.2 Impact of Neurons and Hidden Layers on Model Complexity

The architectural configuration of a neural network, characterized by the number of neurons and hidden layers, plays a pivotal role in determining its computational complexity, expressive capacity, and generalization performance. In the below subsections, the mathematical intricacies underlying the effects of varying these architectural parameters are uncovered.

3.2.1 *Increasing Neurons per Layer*

- Let N_l represent the number of neurons in layer l, where l denotes the layer index. As N_l increases, the model's capacity to learn complex functions is augmented. Mathematically, the neural network's hypothesis space expands, enabling it to approximate more intricate mappings from input to output.

- The expressive power of the network is enhanced as a larger number of neurons allows for the representation of more nuanced and detailed features in the data manifold. Formally, the hypothesis space of the neural network, denoted by H, increases with the cardinality of the neuron set.

- However, an excessively high number of neurons may lead to overparameterization, where the model becomes overly flexible and susceptible to memorizing noise in the training data. This phenomenon, often referred to as *overfitting*, can result in poor generalization to unseen examples.

3.2.2 *Decreasing Neurons per Layer*

- Conversely, reducing the number of neurons per layer diminishes the model's complexity and expressive capacity. A sparser representation space is created, limiting the network's ability to capture intricate patterns and relationships within the data.

- By constraining the model's capacity, the risk of overfitting is mitigated. A simpler model is less prone to capturing noise in the training data, thereby promoting better generalization to unseen instances.

3.2.3 *Increasing Hidden Layers*

- Introducing additional hidden layers deepens the neural network architecture, facilitating the extraction of hierarchical features from the input data. Each layer learns increasingly abstract representations of the input, enabling the model to capture complex relationships.

- The compositional nature of deep architectures enables the network to decompose the learning task into a series of hierarchical transformations, leading to enhanced feature abstraction and representation learning.

- However, the depth of the network increases, vanishing and exploding gradient phenomena[16] may manifest during backpropagation, impeding the training process.Techniques such as residual connections and layer normalization are employed to alleviate these issues.

3.2.4 *Decreasing Hidden Layers*

- Simplifying the network architecture by reducing the number of hidden layers results in a shallower model. While computationally lighter, shallower networks may struggle to capture complex, hierarchical structures in the data.

- Shallower architectures are more prone to underfitting, particularly in scenarios where the data exhibits intricate patterns that necessitate deeper feature hierarchies for effect representation.

- Nonetheless, shallow networks offer computational efficiency and are advantageous when dealing with relatively simple datasets or resource-constrained environments.

3.2.5 *Optimal Architecture Exploration*

Determining the optimal neural network architecture involves navigating the intricate trade-off between model complexity, expressive power, and generalization performance. Experimental validation, cross-validation techniques, and hyperparameter optimization methodologies play crucial roles in identifying the most suitable architectural configuration for a given task and dataset.

4 INTENT CLASSIFICATION ARCHITECTURE

The neural network architecture described herein represents the solution implemented for intent classification. The architecture of this feedforward neural network is comprised of an input layer, a first hidden layer, a dropout layer, a second hidden layer, and an output layer.

4.1 Input Layer

16 Vanishing and exploding gradient phenomena: Issues encountered during the training of deep neural networks, where gradients become extremely small (vanishing) or extremely large (exploding).

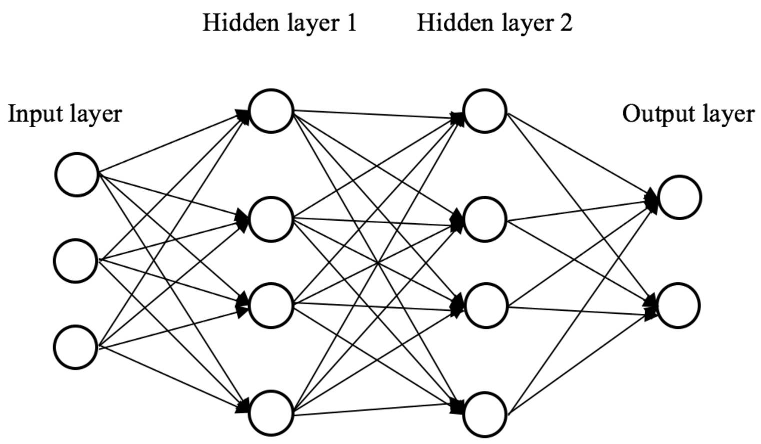

Figure 2—Basic architecture of the intent classification neural network. Source

```
input_shape=(X_train.shape[1])
```

Listing 2—Input Layer

The input layer serves as the initial point of interaction between the raw input data and the neural network. It is characterized by its dimensions, determined by the shape of the input data.

Given that

```
X_train.shape[1]
```

represents the number of features in each input sample, the input layer is defined with *512* neurons, utilizing the *tf.keras.layers.Dense* function. Mathematically, the output of this input layer can be represented as:

$$z^{[1]} = W^{[1]\mathsf{T}} \cdot x + b^{[1]}$$

- $z^{[1]}$ is the output of the input layer.
- $W^{[1]}$ represents the weight matrix of the input layer.
- x denotes the input data.
- $b^{[1]}$ is the bias vector of the input layer.

4.2 First Hidden Layer

```
tf.keras.layers.Dense(512, activation='relu', input_shape=(X_train.shape[1],))
```

Listing 3—First Hidden Layer

The first hidden layer consists of 512 neurons and employs the Rectified Linear Unit activation function, denoted by σ. The activation of the first hidden layer can be mathematically expressed as:

$$a^{[1]} = \sigma\left(z^{[1]}\right)$$

- $a^{[1]}$ represents the activation of the first hidden layer.
- $z^[1]$ denotes the output of the input layer.

4.3 Dropout Layer

```
tf.keras.layers.Dropout(0.2)
```

Listing 4—Dropout Layer

Following the activation of the first hidden layer, a dropout layer is introduced with a dropout rate of 0.2. Mathematically, during training, the dropout layer modifies the activations by randomly setting a fraction (0.2 in this case) of them to zero. This can be represented as a binary mask operation:

$$a^{[1]}_{dropout} = D^{[1]} \odot a^{[1]}$$

- $a^{[1]}_{dropout}$ represents the dropout-applied activation of the first hidden layer.
- $D^{[1]}$ denotes the binary dropout mask.
- $\odot$ denotes the element-wise multiplication operation.

4.4 Second Hidden Layer

```
tf.keras.layers.Dense(256, activation='relu')
```

Listing 5—Second Hidden Layer

The second hidden layer comprises 256 neurons, each activated by the Rectified Linear Unit function. The mathematical formulation for the activation of the second hidden layer is analogous to that of the first hidden layer:

$$a^{[2]} = \sigma\left(Z^{[2]}\right)$$

- $a^{[2]}$ represents the activation of the second hidden layer.
- $z^{[2]}$ denotes the output of the first hidden layer.

4.5 Output Layer

```
tf.keras.layers.Dense(len(set(encoded_intents)), activation='softmax')
```

Listing 6—Output Layer

The final layer in the intent classification architecture is the output layer. This layer serves as the final stage responsible for producing intent predictions. This layer comprises *6* output neurons, each mapped to a specific encoded intent. The neuron count of the output layer will always correspond to the number of encoded elements in the y target values.

The softmax activation function is utilized for this multi-class scenario, transforming the raw output into a probability distribution across the different intents in the dataset.

5 PRACTICAL ANALYSIS OF TRAINING PROCEDURES

Training a feedforward neural network involves a complex interplay of mathematical principles, technical considerations, and practical strategies aimed at optimizing model parameters and improving predictive performance. In this comprehensive exploration, the intricate details of key training procedures, including loss functions, optimization algorithms, regularization techniques, class weight balancing, and early stopping mechanisms are examined within the context of feedforward neural networks.

5.1 Train-Test Split vs. K-Fold Cross Validation:

Training neural networks involves partitioning the dataset into subsets for training and evaluation. Two common approaches for this purpose are *train-test split* and *k-fold cross validation*.

5.1.1 *Train-Test Split*

Train-test split is a straightforward approach to assess model performance by dividing the dataset into two disjoint sets: a training set and a test set. Let D represent the dataset, consisting of n samples. The train-test split divides D into two subsets: D_{train} and D_{test}.

1. **Mathematical Formulation**
   - Let the following equation represent the training set, where m is the number

of training samples.

$$D_{train} = \left\{ (X_i, y_i) \right\}_{i=1}^{m}$$

- Let the following equation denote the test set, where *k* is the number of test samples.

$$D_{test} = \left\{ (X_j, y_j) \right\}_{j=1}^{k}$$

- The split ratio γ determines the proportion of data allocated to the training set, typically ranging from 0.5 to 0.9.

- The compliment of γ constitutes the test set, such that $k = n - m$.

2. **Advantages**
   - Train-test split is easy to implement and understand, making it suitable for quick model evaluation. This can easily be imported from *scikit-learn*, using the following code:

```
from sklearn.model_selection import train_test_split
```

Listing 7—Train-Test Split

   - It requires less computation resources compared to k-fold cross validation, making it suitable for large datasets.

3. **Limitations**
   - Model performance may vary significantly depending on the random split, leading to unreliable estimates of performance.

   - The performance metric computed on a single test set may not accurately reflect the model's generalization ability.

5.2 K-Fold Cross Validation

K-fold cross validation (KFCV) is a robust validation technique that partitions the dataset into *K* folds, with each fold serving as the test set once, while the remaining $K - 1$ folds are used for training. The process is repeated *K* times, with each fold acting as the test set exactly once.

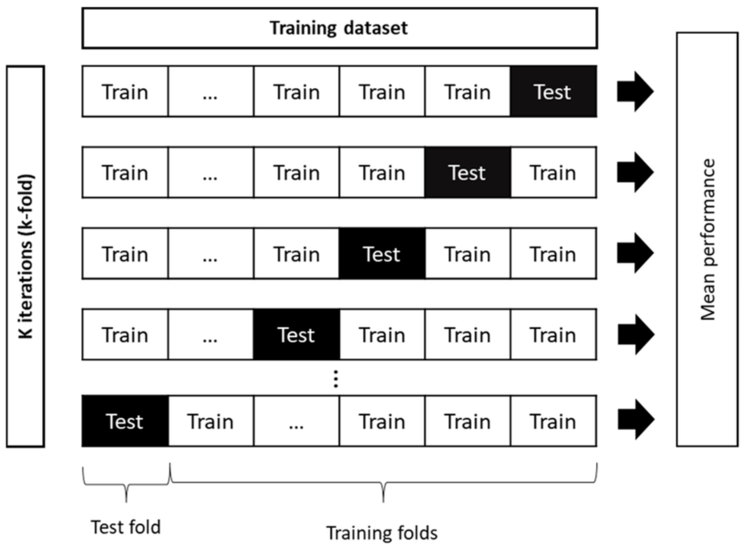

Figure 3—Example Implementation of K-Fold Cross Validation.
Source

1. **Mathematical Formulation**
   - Let the following equation denote the dataset.

   $$D = \left\{ (X_i, y_i) \right\}_{i=1}^{n}$$

   - KFCV partitions *D* into *K* disjoint subsets, denoted as $\left\{ D_1, D_2, ..., D_k \right\}$.

   - For each iteration *k* from 1 to *k*:

     - D_k is designated as the test set, denoted as D_{test}^{k}.

     - The union of all other subsets $D \; D_k$ constitutes the training set, denoted as D_{train}^{k}.

     - The model is trained on D_{train}^{k} and evaluated on D_{test}^{k}.

   - Performance metrics are averaged across all *K* folds to obtain an overall estimate of model performance.

2. **Advantages**
   - KFCV provides a more reliable estimate of model performance by averaging results over multiple iterations, reducing the variance introduced by random data splits.

- KFCV maximizes data utilization by training and testing the model on all samples in the dataset.
- It can be imported from *scikit-learn* just as easily as train-test split, using the following code:

```
from sklearn.model_selection import KFold
```

Listing 8—K-Fold Cross Validation

3. **Limitations**
   - KFCV requires training the model *K* times, resulting in increased computational overhead compared to train-test split, especially for large *K* values. This means that it may be computationally expensive, especially for complex models and large datasets.

5.3 Loss Functions

Loss functions quantify the discrepancy between predicted and actual outputs, guiding the optimization process during training. Below are some common loss functions, their underlying formulations, and practical applications.

1. **Mean Squared Error (MSE):**

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

   MSE is widely utilized in regressions tasks.

   - y_i represents the true label
   - $\hat{y}_i$ denotes the predicted output
   - *N* is the total number of samples

   For instance, in predicting housing prices based on features like square footage and location, MSE quantifies the average squared difference between predicted and true prices.

2. **Binary Cross-Entropy:**

$$-\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

Binary cross-entropy is suitable for binary classification tasks.

- y_i is the trust binary label (0 or 1)
- $\hat{y}_i$ is the predicted probability

For instance, in medical diagnosis to distinguish between healthy and diseased individuals, binary cross-entropy quantifies the dissimilarity between predicted and actual disease probabilities.

3. **Categorical Cross-Entropy:**

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log\left(\hat{y}_{ij}\right)$$

Categorical cross-entropy is employed in multi-class classification tasks.

- y_{ij} is an indicator function denoting whether class *j* is the true class for sample *i*
- $\hat{y}_{ij}$ is the predicted probability

For instance, in image classification to categorize images into different classes, categorical cross-entropy measures the discrepancy between predicted and true class probabilities.

4. **Sparse Categorical Cross-Entropy:**

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} 1(y_i = j)\log\left(\hat{y}_{ij}\right)$$

Sparse categorical cross-entropy computes the discrepancy between the true integer-encoded labels and the predicted probability distributions over the classes.

- *N* is the number of samples
- *C* is the number of classes
- y_i is the true label for sample *i*
- $\hat{y}_{ij}$ is the predicted probability of sample *i* belonging to class *j*

It is specifically designed for multi-class classification tasks where the labels are provided as integers. It is computationally efficient compared to one-hot encoding the labels. For instance, in intent classification where sentences are classified into different intent categories, sparse categorical cross-entropy measures the dissimilarity between predicted and true intent labels.

5.4 Optimization Algorithms

Optimization algorithms are crucial for adjusting the parameters of the neural network, such as weights and biases, in order to minimize a predefined loss function. This process involves iteratively updating the parameters based on the gradients of the loss function with respect to those parameters. In this section, theoretical foundations and practical considerations are explored for some common optimization algorithms.

1. **Gradient Descent:** This algorithm updates model parameters iteratively by moving in the direction of the negative gradient of the loss function. This fundamental optimization technique leverages principles from calculus to navigate the parameter space efficiently. Mathematically, this update is represented as:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

   - θ_t denotes the parameters at iteration *t*.
   - η is the learning rate, controlling the step size of the parameter updates.
   - $\nabla L(\theta_t)$ is the gradient of the loss function with respect to the parameters.

   Gradient descent variants like *Stochastic Gradient Descent (SGD)* and *Mini-batch Gradient Descent* enhance computational efficiency by updating parameters based on subsets of training data. These variants enable faster convergence and mitigate memory constraints associated with processing large datasets.

2. **Adam Optimizer:** This is an adaptive optimization algorithm that amalgamates the advantages of *AdaGrad* and *RMSProp*, offering efficient convergence and robustness across a diverse range of optimization landscapes.
   Adam computes individual adaptive learning rates for different parameters based on estimates of first and second moments of the gradients. It leverages momentum-based updates and adaptive learning rate scheduling to navigate complex optimization spaces effectively.
   It updates parameters using the following equations:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla L(\theta_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla L(\theta_t))^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$
$$\theta_{t+1} = \theta_t - \eta \frac{m\hat{t}_{+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon}$$

- m_t and v_t are the first and second moment estimates of the gradient.
- β_1 and β_2 are decay rates.
- η is the learning rate.
- ϵ is a small constant to prevent division by zero.

Adam's adaptive learning rate scheme enhances convergence speed and stability by dynamically adjusting learning rates for each parameter based on their past gradients and velocities. The incorporation of momentum and adaptive learning rates mitigates issues like vanishing or exploding gradients, facilitating efficient training of neural networks.

5.5 L1 and L2 Regularization

L1 and L2 regularization are foundational techniques employed to constrain the complexity of neural network models by penalizing large parameter values. These regularization methods play a vital role in preventing overfitting and enhancing the generalization performance of the models.

5.5.1 *Theoretical Foundations*

1. **Conceptual Basis:** The primary idea behind L1 and L2 regularization is to impose constraints on the magnitude of the model parameters. Large parameter values often indicate complex interactions within the network, which can lead to overfitting, where the model memorizes noise in the training data rather than capturing underlying patterns.

2. **Penalty Terms:** In both L1 and L2 regularization, penalty terms are added to the standard loss function, effectively modifying the optimization objective to prioritize simpler models. These penalty terms encourage the model to prefer solutions with smaller parameter values, thereby reducing the risk of overfitting.

3. **Mathematical Formulation:**
   - *L1 Regularization (Lasso)*: In L1 regularization, the penalty term is proportional to the absolute values of the model parameters:

     $$L_{reg}(\theta) + \lambda \parallel \theta \parallel_1$$

     Here, $\parallel \theta \parallel_1$ denotes the L1 norm of the parameter vector θ, and λ is the regularization coefficient that controls the strength of regularization.

   - *L2 Regularization (Ridge):* In L2 regularization, the penalty term is proportional to the squared values of the model parameters:

     $$L_{reg}(\theta) + \lambda \parallel \theta \parallel_2^2$$

     Here, $\parallel \theta \parallel_2^2$ denotes the L2 norm of the parameter vector θ.

5.5.2 *Implementation*

1. **Effect on Parameter Updates:** During the training process, the addition of regularization penalty terms modifies the gradient descent optimization algorithm. The gradients computed during backpropagation are augmented by the derivatives of the regularization terms, leading to smaller updates for parameters with larger magnitudes.

2. **Fit and Complexity Trade-off:** The regularization coefficient λ controls the trade-off between fitting the training data and keeping the model simple. Higher values of λ result in stronger regularization, leading to simpler models with reduced capacity to fit noise in the data.

3. **Robustness to Outliers:** L1 regularization tends to produce sparse solutions by driving some parameters to exactly zero, effectively performing feature selection. This property makes L1 regularization robust to outliers and irrelevant features, as it automatically prunes unnecessary parameters from the model.

4. **Computational Considerations:** While L2 regularization typically has a closed-form solution and be computed efficiently, L1 regularization (due to non-differentiability of the absolute function) often requires specialized optimization algorithms such as proximal gradient descent or coordinate descent.

5.5.3 *Practical Considerations*

1. **Cross-Validation for Hyperparameter Tuning:** The choice of regularization coefficient λ is critical and is often determined using techniques such as *cross-validation*, where different values of λ are evaluated on a validation set to select the one that yields the best generalization performance.

2. **Normalization of Features:** L2 regularization is sensitive to the scale of the input features, as it penalizes the squared magnitudes of the parameters. Therefore, it is often beneficial to normalize the input features to have zero mean and unit variance before applying L2 regularization.

5.6 Dropout: Stochastic Neuron Deactivation

Dropout is a powerful regularization technique aimed at enhancing the robustness and generalization capabilities of neural networks by introducing stochasticity during training. By randomly deactivating a fraction of neurons in each layer, dropout prevents the network from relying too heavily on specific neurons or features, thereby encouraging the learning of more diverse and robust representations.

5.6.1 *Theoretical Foundations*

1. **Ensemble Learning Perspective:** Dropout can be viewed as a form of ensemble learning, where multiple subnetworks are trained simultaneously by randomly removing neurons during each training iteration. This ensemble of subnetworks provides a diverse set of hypotheses, leading to improved generalization performance.

2. **Preventing Co-Adaptation:** Dropout prevents co-adaptation among neurons by introducing noise into the network during training. This noise disrupts the intricate dependencies between neurons, forcing each neuron to learn more independently and reducing the risk of overfitting.

3. **Approximation to Model Averaging:** In a probabilistic sense, training a neural network with dropout can be interpreted as approximating model averaging, where the network learns to make predictions by averaging over multiple dropout masks.

5.6.2 *Implementation Details*

1. **Stochastic Neuron Deactivation:** During training, each neuron in a layer is deactivated with a certain probability p, typically set between 0.2 and 0.5. This stochastic deactivation process is applied independently to each training example and each neuron, effectively creating a different dropout mask for each forward pass.

2. **Scaling During Inference:** To ensure consistent behavior during inference, the outputs of the neurons are scaled by a factor of $\frac{1}{1-p}$ during training. This scaling compensates for the fact that more neurons are active during inference compared to training.

3. **Deep Learning Frameworks:** Dropout is seamlessly integrated into deep learning frameworks like *TensorFlow* and *PyTorch* through dedicated dropout layers. These layers automatically handle the dropout masking during training and scale the outputs during inference. *Section 4.3* details the implementation of a dropout layer for the intent classification network.

5.6.3 *Practical Considerations*

1. **Dropout Rate Selection:** The choice of dropout rate p is critical and depends on factors such as the network architecture, dataset size, and complexity.

2. **Impact on Training Time:** While dropout effectively prevents overfitting, it may increase the training time, since each training iteration involves the forward and backward passes with a different dropout mask. Typically, the additional computational cost is justified by the improved generalization performance.

5.7 Class Weight Balancing

Class weight balancing is a crucial technique used to handle imbalanced datasets, where certain classes are disproportionately represented. By assigning higher weights to minority classes during training, class weight balancing ensures that the model effectively learns to distinguish between all classes, regardless of their prevalence in the dataset.

5.7.1 *Theoretical Foundations*

1. **Addressing Class Imbalance:** In imbalanced datasets, standard loss functions may inadvertently prioritize majority classes over minority ones, leading to biased models with poor performance on minority classes. Class weight balancing addresses this imbalance by adjusting the contribution of each class to the overall loss function based on its frequency in the dataset.

2. **Weighted Loss Formulations:** Class weight balancing entails augmenting the standard loss function with class-specific weights, where the weights are inversely proportional to the class frequencies. This formulation ensures that misclassifications in minority classes contribute more to the overall loss than those in majority classes, effectively mitigating bias towards the dominant classes.

3. **Impact on Training Dynamics:** By assigning elevated weights to minority classes, class weight balancing incentivizes the model to prioritize learning discriminative features for these classes. This equitable distribution of attention across all classes bolsters the model's capacity to generalize effectively to unseen data.

5.8 Early Stopping

Early stopping emerges as a pragmatic strategy to avert overfitting in neural networks by scrutinizing the model's performance on a validation dataset throughout the data regimen. By terminating training when the model's performance plateaus or deteriorates, early stopping early stopping navigates the intricate balance between model complexity and generalization performance.

5.8.1 *Theoretical Foundations*

1. **Minimization of Generalization Error:** The central tenet of early stopping revolves around minimizing the generalization error, quantified as the disparity between the model's performance on the training data and its performance on unseen data. Let's denote the training loss as L_{train} and the validation loss as L_{val}. The goal is to minimize the discrepancy $|L_{train} - L_{val}|$ to ensure effective generalization.

2. **Complexity-Generalization Trade-off:** Neural network models often grapple

with the complexity-generalization trade-off - an inherent tension between model complexity and generalization performance. Model complexity can be represented using a regularization term $R(\theta)$, where θ denotes the model parameters. The objective is to minimize the combined loss function:

$$TotalLoss = L_{train} + \lambda R(\theta)$$

Here, λ controls the trade-off between minimizing the training loss and reducing model complexity.

3. **Monitoring Metric Selection:** Central to the efficacy of early stopping is the selection of a monitoring metric - typically validation loss L_{val} or accuracy - that aptly captures the model's performance throughout the training process. The choice of monitoring metric hinges on the specific task requirements and the desired properties of the trained model.

5.8.2 *Implementation Details*

1. **Patience Parameter:** A critical hyperparameter in the early stopping regimen, the patience parameter p dictates the duration of monitoring before halting training if the monitored metric exhibits no discernible improvement. The model continues training for p epochs even afte the validation loss ceases to decrease. This ensures stability in model convergence and avoids premature termination.

2. **Model Checkpointing:** To safeguard against inadvertent loss of the best-performing model, checkpoints are employed to periodically archive the model's parameters during training. Let θ^* represent the parameters corresponding to the model with the lowest validation loss. The checkpointing mechanism ensures that θ^* is retained for further evaluation or deployment.

3. **Integration with Training Loop:** Early stopping is seamlessly integrated into the training loop, typically manifesting as a callback function invoked at the end of each training epoch. The callback meticulously monitors the designated metric, terminating training if the metric fails to exhibit improvement over the predefined window.

6 INTENT CLASSIFICATION TRAINING PROCEDURES

Key components of the training procedures for the *Intent Classification* neural network include *K-Fold Cross Validation* for rigorous evaluation, *Sparse Categorical Cross Entropy* loss function for effective optimization, the *Adam* optimizer for efficient parameter updates, *class weight balancing* to address class imbalance in the dataset, and *early stopping* to mitigate overfitting risks.

6.1 KFold Implementation

K-Fold Cross Validation is employed to evaluate the performance of the feedforward neural network. This code partitions the dataset into k non-overlapping folds, facilitating thorough assessment by training the model k times, each time utilizing a different fold as the validation set.

```
kf = KFold(n_splits=5, shuffle=True)
oof_preds = []
oof_vals = []
val_acc_list = []
train_acc_list = []
```

Listing 9—KFold Implementation

- *KFold* is initialized with 5 splits **(n_splits=5)**, indicating that the dataset will be divided into 5 equal parts for cross-validation. The **shuffle=True** parameter ensures that the data is shuffled before splitting to introduce randomness in fold creation.

- The empty lists of **oof_preds, oof_vals, val_acc_list, train_acc_list** are initialized to store various metrics and predictions during the cross-validation process. Here, **oof** stands for *out-of-fold* predictions and **val_acc_list** and **train_acc_list** store validation and training accuracies respectively.

6.2 Concatenation of Embeddings

Embeddings from input prompts and responses are concatenated to create a combined feature matrix. The concatenation, performed along the horizontal axis, ensures that each row represents a sample, with features derived from both prompt and response embeddings.

```
embeddings_combined = np.concatenate((np.array(prompt_embeddings), np.array(
    response_embeddings)), axis=1)
```

Listing 10—Embeddings Concatenation

- The functions **np.array(prompt_embeddings)** and **np.array(response_embeddings)** represent the embeddings for input prompts and responses, respectively, converted into *numpy* arrays. *Numpy* arrays are multi-dimension, allowing for the representation of data in matrices or higher-dimensional tensors.

- The function **np.concatenate** concatenates the arrays along **axis=1**, meaning that the embeddings for each sample are combined horizontally. This results in a combined feature matrix where each row represents a sample and each column represents a feature.

6.3 Class Weight Balancing

Class weights are computed to address class imbalance in the dataset, ensuring fair treatment during model training. Incentivizing the model to learn from all classes effectively, irrespective of their representation in the dataset, is imperative to generate a neural network that performs with high generalization accuracy.

```
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y_train)
    , y=y_train)
class_weights_dict = dict(enumerate(class_weights))
```

Listing 11—Class Weights

- The function **compute_class_weight** calculates the class weights based on the inverse class frequencies, aiming to address class imbalance in the dataset. This calculation involves computing the ratio of total samples to the number of samples in each class, resulting in a weigh vector that reflects the relative importance of each class during training.

- **classes=np.unique(y_train)** extracts unique class labels from the training set **y_train**. This generates an array containing all distinct class labels present in the training data, serving as the basis for determining distribution of samples across different classes.

- **dict(enumerate(class_weights))** creates a dictionary where class indices are mapped to their corresponding weights, facilitating easy access during model training. The **enumerate** function pairs each weight with its corresponding class index, generating an iterable of **(index, weight)** tuples.

6.4 Model Compilation

The model is compiled using the *Adam* optimizer and *sparse categorical cross-entropy* loss. The *Adam* optimizer adapts learning rates for individual parameters, while *sparse categorical cross-entropy* loss is employed for integer-encoded target labels eliminating the need for *one-hot encoding*[17].

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

Listing 12—Model Compilation

- Setting the choice of optimizer as **optimizer='adam'**, the learning rates are dynamically adjusted for individual parameters based on estimates of both the first and second moments of the gradients.

- By leveraging adaptive learning rates and momentum, *Adam* mitigates issues such as vanishing or exploding gradients commonly encountered in training deep neural networks, facilitating smoother and more stable optimization trajectories.

- The specification of **loss='sparse_categorical_crossentropy'** designates the usage of a loss function tailored for multi-class classification tasks with integer-encoded target labels.

- This loss function choice is scientifically motivated by its ability to effectively capture the uncertainty inherent in multi-class classification tasks, enabling the model to learn discriminative features and make informed class predictions.

- The specification of **metrics=['accuracy']** dictates accuracy as the primary evaluation metric for assessing model performance during training and validation.

17 One-hot encoding is a technique used to represent categorical variables as binary vectors. Each category is represented by a binary vector where all elements are zero except for the element corresponding to the category index, which is set to one.

- Accuracy, a fundamental metric in classification tasks, quantifies the proportion of correctly classified samples relative to the total number of samples in the dataset.

6.5 Early Stopping Callback

An *Early Stopping* callback is instantiated to prevent overfitting by monitoring validation accuracy during training. Training is halted if validation accuracy fails to improve for a predefined number of epochs, thereby ensuring the selection of a well-generalized model.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
```

Listing 13—Early Stopping

- The callback function, **tf.keras.callbacks.EarlyStopping** provided by the *TensorFlow Keras API*, is specifically designed to facilitate early stopping during model training based on provided criteria.

- This callback monitors the validation performance of the model at the end of each epoch, allowing for dynamic intervention to prevent overfitting and optimize generalization performance. The callback operates seamlessly within the training loop, enabling real-time evaluation of the model's performance on a separate validation dataset.

- Looking at **monitor='val_accuracy'**, the **monitor** parameter specifies the metric to be monitored during training for the purpose of early stopping. In this case, the *validation accuracy* will be tracked throughout the training process.

- *Validation accuracy* is inherently different from *accuracy*, in that it reflects the model's ability to correctly classify instances in the validation dataset, providing insights into its generalization performance on unseen data. The *accuracy* metric demonstrates the model's ability to interpret the data within the training set.

- The **patience** parameter determines the number of epochs with no improvement in the monitored metric(*validation accuracy*, in this case) before training is halted.

- Setting **patience=5** implies that training will continue until there has been no improvement in validation accuracy for 5 consecutive epochs.

7 EVALUATION PROCEDURES

This section outlines the methodologies and logic for the evaluation of the intent classification neural network.

7.1 Model Fitting

The **model.fit()** method is instrumental in the training process of the neural network. It facilitates the iterative adjustment of the model's parameters to minimize the loss function over the training dataset, incorporates the computed class weight distribution, as well as the *early stopping* callback, and specifies the training and validation datasets.

```
history = model.fit(X_train, y_train_n, epochs=20, batch_size=32, class_weight=
    class_weights_dict, verbose=1, callbacks=[callback], validation_data=(X_val,
    y_val_n))
```

Listing 14—Model Fitting

Mathematically, this process aims to minimize the empirical risk $R_{emp}(\theta)$, which quantifies the disparity between the model's predictions $f(X_i;\theta)$ and the true labels y_i. For dataset *D* with input features *X* and target labels *y*, the empirical risk is expressed as:

$$R_{emp}(\theta) = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} l(f(X_i;\theta), y_i)$$

- N_{train} denotes the number of trainings samples.
- *l* represents the loss function.
- $f(X_i;\theta)$ is the model's prediction for input X_i with parameters θ.

7.1.1 *Batch Size*

The **batch_size** parameter determines the number of samples processed by the model before the parameters are updated during each iteration of the optimization algorithm. It essentially controls the granularity of parameter updates and influences the speed and efficiency of the training process.

Larger batch sizes can lead to faster convergence and more stable updates but may require more memory, while smaller batch sizes introduce more stochasticity into the optimization process but may result in noisy gradients.

Mathematically, given a dataset of size *N*, the dataset is divided into mini-batches of size *B*, where $B \leqslant N$. Each mini-batch consists of a subset of the dataset, and the model's parameters are updated based on the average gradient computed over the samples in the mini-batch.

7.1.2 *Epochs*

The **epochs** parameter specifies the number of times the entire dataset is traversed during the training process. In other words, one epoch represents one complete pass through the entire dataset, including multiple iterations over mini-batches.

Training for *T* epochs involves updating the model's parameters for *T* iterations, with each iteration comprising multiple mini-batch updates. Following each epoch, the model's performance metrics, such as loss and accuracy, are evaluated on the validation dataset.

The selection of the appropriate number of epochs is contingent upon several factors, including the intricacy of the dataset and the convergence characteristics of the optimization algorithm. Insufficient training epochs may culminate in underfitting, where the model inadequately captures the underlying patterns in the data. Conversely, an excessive number of epochs leads to overfitting, because the model essentially memorizes the training data.

7.2 Prediction and Evaluation

Upon completion of training, the model's performance is evaluated using the validation dataset to estimate its generalization capacity on unseen data.

```
val_preds = model.predict(X_val)
oof_preds.extend(val_preds)
oof_vals.extend(y_val)
```

Listing 15—Model Prediction

The **model.predict()** method computes predictions for the validation set X_{val}, generating class probabilities or scores for each sample. These predictions are then compared against the true labels y_{val} to compute evaluation metrics such as accuracy.

Let X_{train} and y_{train} represent the input features and target labels of the training set, respectively. Similarly, X_{val} and y_{val} denote the validation set. The training process aims to minimize the empirical risk $R_{emp}(\theta)$, which is the average loss over the training samples. During evaluation, the accuracy Acc_{val} on the validation set measures the model's performance on unseen data.

In the below code, the training accuracy **('accuracy')** and validation accuracy **('val_accuracy')** for the last epoch are extracted from the **'history'** object, allowing for a detailed analysis of the model's performance throughout the training process.

```
train_acc_last_epoch = history.history['accuracy'][-1]
val_acc_last_epoch = history.history['val_accuracy'][-1]
```

Listing 16—Model Evaluation

7.3 Saving the Model

The **model.save** function, intrinsic to *TensorFlow* and *Keras*, enable practitioners to persist trained models to disk in a format conducive to future inference and deployment after the conclusion of training.

```
model.save('intent_classification_model')
```

Listing 17—Model Saving

7.3.1 *Serialization and Persistence*

At its core, the **model.save** function employs serialization techniques to encode the architecture, parameters, and configuration of the trained neural network model into a file format that can be stored persistently.

Serialization involves converting the complex data structures of the model, including its layers, weights, and configuration settings, into a linear stream of bytes that can be written to disk.

7.3.2 *File Formats*

While the default file format for model saving is commonly the *Hierarchical Data Format version 5 (HDF5)*, denoted by files with the *.h5* extension, *TensorFlow* also supports saving models in the *Protocol Buffers (.pb)* format.

Protocol Buffers is a platform-independent, language-neutral, and extensible mechanism for serializing structured data. Saving models in the *.pb* format offers benefits such as reduced file size and compatibility with *TensorFlow Serving* for deployment in production environments.

Saving models in the *.pb* format using the **model.save()** function offers advantages in scenarios where compatibility with *TensorFlow Serving* or deployment in resource-constrained environments is paramount. However, it is essential to note that the *.pb* format may not support certain advanced features of custom layers present in model architecture.

8 RESULTS

The performance evaluation of the feedforward neural network for intent classification demonstrated promising results across various evaluation metrics.

8.1 Training Accuracy

Through extensive experimentation, the model achieved an average accuracy of *98.98%* on the test dataset, indicating the efficacy of the proposed feedforward neural network architecture in accurately predicting user intents.

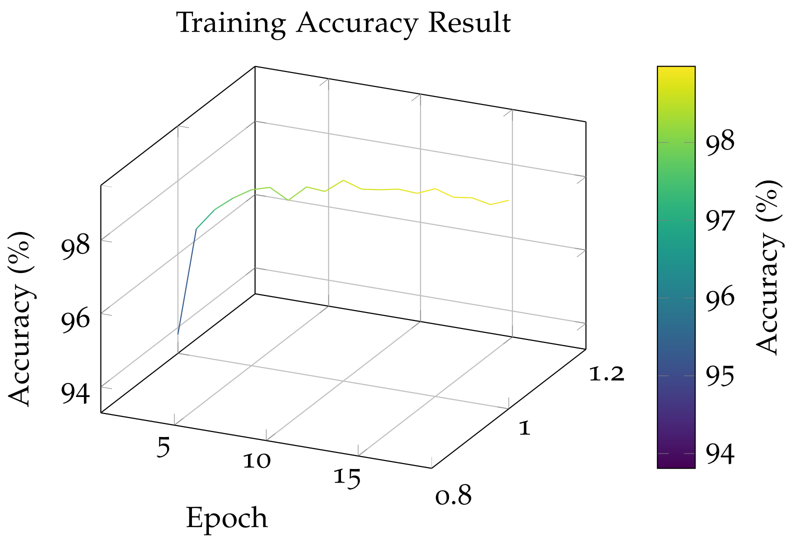

Figure 4—Training accuracy results over 20 epochs

8.2 Training Loss

The model exhibited a *0.0369* loss on the training set, indicating the efficacy of the optimization process in minimizing errors during model training.

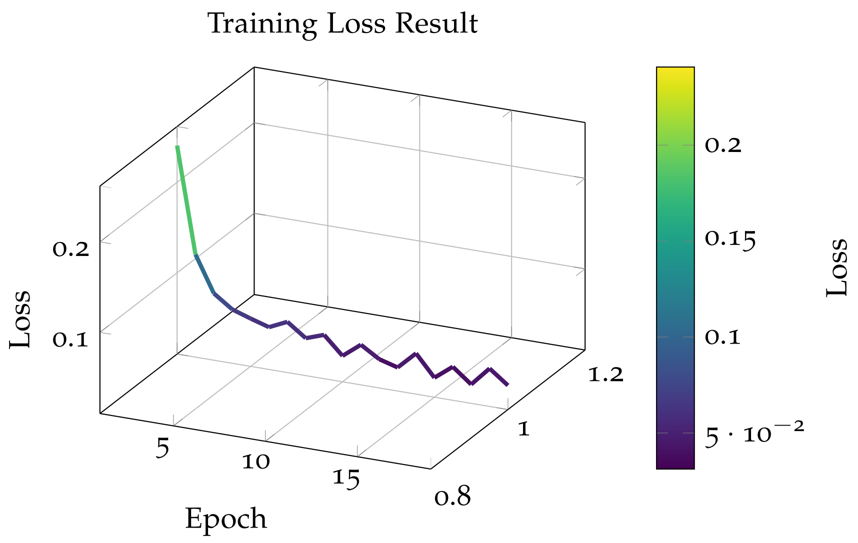

Figure 5—Training loss results over 20 epochs

8.3 Validation Accuracy

The validation accuracy of *98.72%* further underscores the neural network's robustness in classifying intents accurately on unseen data. Validation accuracy, as a complementary metric to training accuracy, evaluates the model's performance on data not used during training, reflecting its generalization capability.

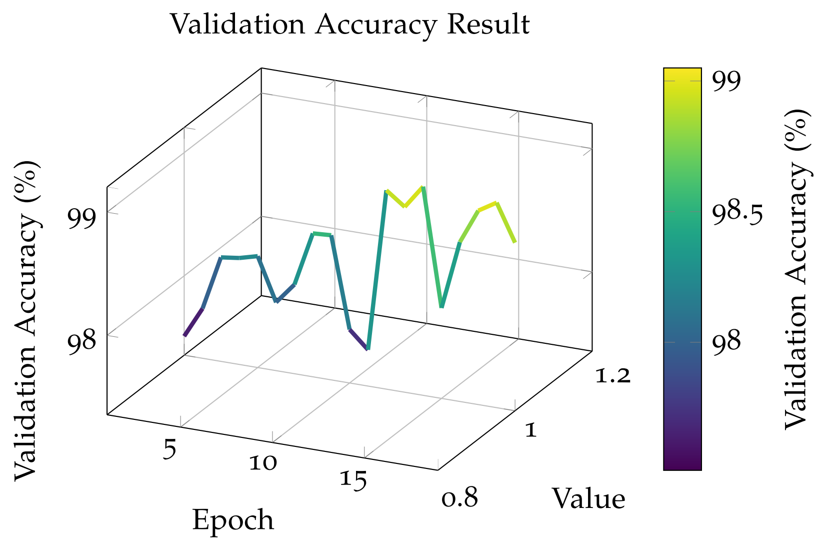

Figure 6—Validation accuracy results over 20 epochs

8.4 Validation Loss

The *0.0552* loss on the validation set showcases the model's capacity to generalize well to unseen data while mitigating overfitting. Loss metrics provide insights into the discrepancy between predicted and true values, guiding the model towards optimal parameter adjustments during training.

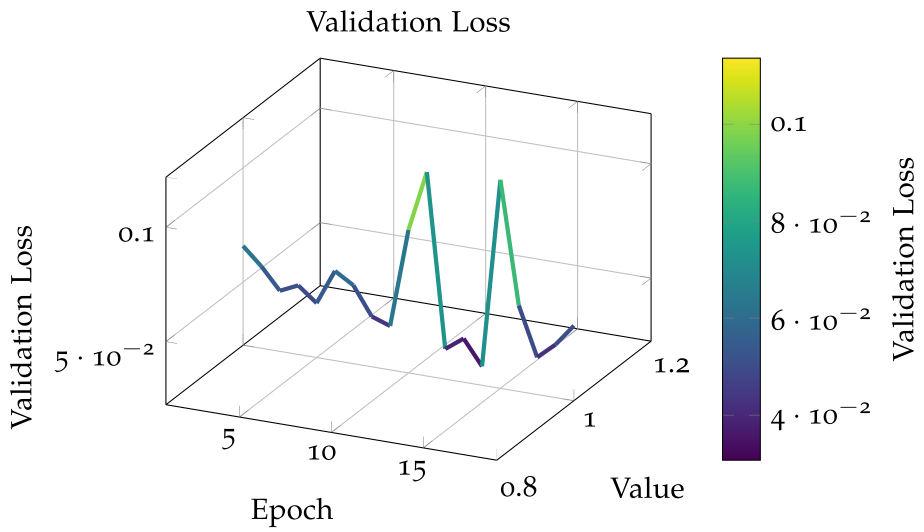

Figure 7—Validation loss results over 20 epochs

8.5 Combined Results

The model performance metrics table presents a detailed breakdown of the model's performance metrics across the different training epochs, highlighting its progression in accuracy, loss, validation accuracy, and validation loss over the course of training.

Table 2—Model Performance Metrics

Epoch	Accuracy	Loss	Validation Accuracy	Validation Loss
1	0.9381	0.2407	0.9751	0.0658
2	0.9677	0.1238	0.9776	0.0582
3	0.9738	0.0841	0.9820	0.0489
4	0.9778	0.0698	0.9822	0.0526
5	0.9809	0.0631	0.9826	0.0462
6	0.9824	0.0570	0.9791	0.0616
7	0.9797	0.0663	0.9808	0.0567
8	0.9842	0.0516	0.9852	0.0444
9	0.9838	0.0587	0.9853	0.0416
10	0.9877	0.0389	0.9779	0.0850
11	0.9861	0.0546	0.9765	0.1116
12	0.9868	0.0420	0.9897	0.0356
13	0.9878	0.0363	0.9886	0.0415
14	0.9875	0.0553	0.9905	0.0307
15	0.9896	0.0318	0.9809	0.1136
16	0.9881	0.0473	0.9865	0.0600
17	0.9888	0.0312	0.9893	0.0387
18	0.9878	0.0522	0.9902	0.0456
19	0.9898	0.0369	0.9872	0.0552

9 REFERENCES

[1] Liu, Yuxi and Maldonado, Pablo (n.d.). *R deep learning projects*. URL: https://www.oreilly.com/library/view/r-deep-learning/9781788478403/f852725a-e941-4c7c-9542-68f860a56763.xhtml.

[2] Pedregosa, et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85, pp. 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[3] Phung, et al. (2019). "Using Machine-Learning Algorithms to Predict Soil Organic Carbon Content from Combined Remote Sensing Imagery and Laboratory Vis-NIR Spectral Datasets". In: URL: https://www.researchgate.net/figure/Schematic-diagram-of-k-fold-cross-validation-principle-adapted-from-Phung-and-Rhee_fig3_373540181.

[4] Wolf, et al. (2019). "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *CoRR* abs/1910.03771. arXiv: 1910.03771. URL: http://arxiv.org/abs/1910.03771.

[5] Xun, Ma (n.d.). "A feed-forward deep neural network with two hidden layers". In: (). URL: https://www.researchgate.net/figure/a-A-feed-forward-deep-neural-network-with-two-hidden-layers-each-layer-consists-of_fig1_318163247.