CSI5180 - Virtual Assistants - Course Project

Intent Detection using Deep Learning Techniques

Paul Mvula - 300169684

April 2021

Abstract

Due to the rapid technological advances made over the years, more people are changing their way of living from traditional to more electronic, and conversational interfaces, i.e., chatbots, virtual assistants, . . . , have gained more popularity and more Natural Language Processing (NLP) and Natural Language Understanding (NLU) services for easy creation of conversational interfaces have been made available each with a set of particular features. This project focuses on the intent detection (classification), a form of NLP, using BERT embeddings and Deep Neural Networks (DNN) classifiers on three English benchmark data sets, ChatBot, AskUbuntu and WebApps. On the three relatively small data sets of 2/5/8 intents, accuracies of 98.11%/89.91%/67.79% were achieved, respectively, with attention masks proving to be useful as inputs for the BERT model on the WebApps data set.

Keywords— Deep Learning, intent detection; BERT embeddings; CNN; hyper-parameter optimization; English, NLU

1 Introduction

Over the years, chatbots and virtual assistants (VA) like Amazon Alexa, Microsoft Cortana, Apple Siri, Google Assistant, have gained more popularity and have become integral parts of our daily lives [8]. The basic architecture of a VA is usually composed of 6 modules: wake word detection, speech to text conversion (Automatic Speech Recognition (ASR)), Intent Detection (NLU), Action, Answer Generation (Natural Language Generation (NLG)) and Speech Synthesis (Text-To-Speech (TTS)). In this project, we focus on the intent detection (NLU) module, because after the speech has been converted to text in the previous modules, it is an important step that determines what the user intends (wants) by initiating the request to the VA or Chatbot.

As the Intent Detection (ID) task comes after the ASR task, in VAs, the task is simply a text classification task that can be solved using Machine Learning (ML), where a model is trained on a set of training samples and when it satisfies the needs of the developer on some defined classification metric, it can be used to make predictions for unseen samples, i.e., questions from users.

In this project, we will focus on [5], and implement their idea. The authors of the article focus on solving the intent detection problem using DNN and a Python library, Hyperas [7] , to automatically optimize hyperparameters in Keras models. We will only use the English data sets as they have not made the additional data sets they used, i.e., Estonian, Latvian, Lithuanian and Russian. Experiments are carried out in the Python programming language with Keras and TensorFlow in Jupyter notebooks. The code (notebooks) and data sets created in this project have been uploaded to GitHub and are available at `https://github.com/womega/CSI5180_Project`.

2 Data sets

Our experiments are carried out on two data corpora built by Braun et. al, [2]: chatbot corpus and StackExchange corpus. The chatbot corpus consists of questions gathered by a Telegram chatbot answering questions about public transport connections and the StackExchange Corpus consists of data from the ask ubuntu platform and Web Applications platform. Before proceeding with building the baseline models, we divided and converted the original json files to three comma separated values style-sheets. For each data set, a train, validation and test file were created with the data distribution shown in the Tables 1, 2 & 3. We also kept the same data distribution as in [5] and [2].

We can notice from these tables that:

1. The data sets are relatively small.
2. The data sets are imbalanced, i.e., the distribution of classes is unequal in the training sets.
3. There are no samples for the "None" class in the AskUbuntu validation set.
4. There are no samples for several classes in the validation and testing sets in the WebApps data set.

Intent	Training	Validation	Testing	Σ
Software Recommendation	12	5	40	57
Shutdown Computer	10	3	14	27
Make Update	9	1	37	47
Setup Printer	8	2	13	23
None	3	-	5	8
Σ	42	11	109	162

Table 1: AskUbuntu data distribution

3 Feature Extraction

A Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and

Intent	Training	Validation	Testing	Σ
FindConnection	49	8	71	128
DepartureTime	31	12	35	78
Σ	80	20	106	206

Table 2: Chatbot data distribution

Intent	Training	Validation	Testing	Σ
Delete Account	7	-	10	17
Filter Spam	4	2	14	20
Sync Accounts	3	-	6	9
Change Password	2	-	6	8
Export Data	2	-	3	5
Download Video	1	-	-	1
Find Alternative	3	4	16	23
None	2	-	4	6
Σ	24	6	59	89

Table 3: WebApps data distribution

right halves of Figure 1, more details can be found in [5] & [11].

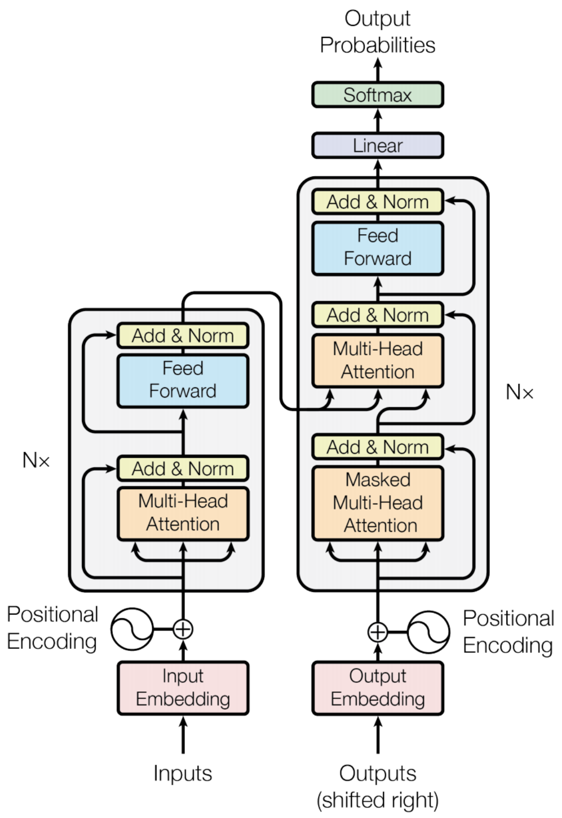
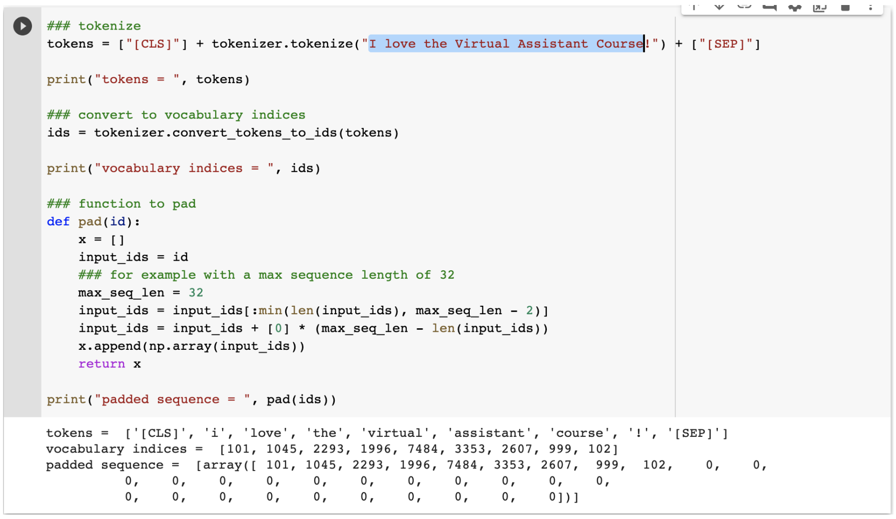


Figure 1: The Transformer – Model Architecture

Before extracting the features, i.e., feeding vectors to the BERT model for feature extraction, we converted the raw texts in the data sets into vectors by following these 3 steps:

1. Tokenize the text using the BERT tokenizer.
2. Convert the sequence of tokens into numbers,i.e., vocabulary indices, with the BERT tokenizer.
3. Pad the sequences (keep the same sequence length). BERT requires the input arrays to be the same size, therefore for sequences to have the same length, we defined a function to pad them in the fine-tuning step, the padding length is calculated by taking the minimum between the longest text and the max sequence length parameter in each data set.

An example of the above steps on the sentence ”I love the Virtual Assistant Course” is shown in Fig. 2. Because this is a single sentence, we default the maximum sequence length to 32, as BERT maximum sequence length is 512.



```
### tokenize
tokens = ["[CLS]"] + tokenizer.tokenize("I love the Virtual Assistant Course!") + ["[SEP]"]

print("tokens = ", tokens)

### convert to vocabulary indices
ids = tokenizer.convert_tokens_to_ids(tokens)

print("vocabulary indices = ", ids)

### function to pad
def pad(id):
    x = []
    input_ids = id
    ### for example with a max sequence length of 32
    max_seq_len = 32
    input_ids = input_ids[:min(len(input_ids), max_seq_len - 2)]
    input_ids = input_ids + [0] * (max_seq_len - len(input_ids))
    x.append(np.array(input_ids))
    return x

print("padded sequence = ", pad(ids))
```

```
tokens =  ['[CLS]', 'i', 'love', 'the', 'virtual', 'assistant', 'course', '!', '[SEP]']
vocabulary indices =  [101, 1045, 2293, 1996, 7484, 3353, 2607, 999, 102]
padded sequence =  [array([ 101, 1045, 2293, 1996, 7484, 3353, 2607,  999,  102,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0])]
```

Figure 2: Example of vectorization steps

The tokenizer model simply creates a fixed-size vocabulary of individual characters, subwords, and words fitting the language data, i.e., English. As the BERT model's vocabulary limit size is 30,000, the WordPiece model [9] generates a vocabulary containing all English characters plus the 30,000 most common words and subwords found in the English language corpus the model is trained on. The vocabulary basically consists of four things [3]:

1. Whole words.
2. Subwords occuring at the front of a word or in isolation.
3. Subwords not at the front of a word, which are preceded by '##' to denote this case.
4. Individual characters.

4 Baseline Models

After extracting the 768 features with BERT, we created baseline models for each of the data sets to make improvements on. The outputs of the BERT tokenizer are fed to the BERT encoder, and the pooled_outputs, which represent the embedding of each input sequence as a whole, of the BERT encoder are fed to a Dense layer with 768 neurons, with a ReLU activation function, whose outputs are then dropped out with a frequency of 0.1 before the last dense layer does the classification.

The architecture of the baseline model for the "askubuntu" is shown in Figure 3, the webapps and chatbot models also follow the same architecture just the last dense layers outputs are (None, 8) and (None, 2), respectively, which corresponds to the number of classes for each task.

For pure evaluation and as a default, we used "sparse categorical crossentropy" loss function for askubuntu and webapps as they both are multiclass classification tasks

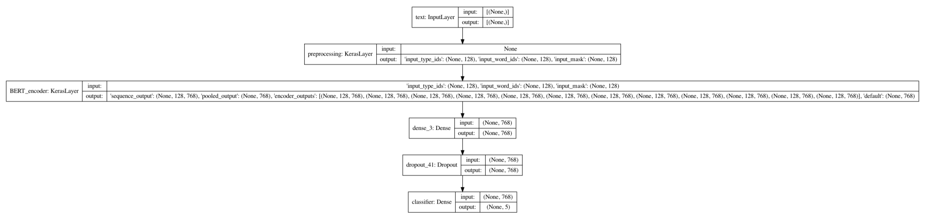

text: InputLayer
input: [(None,)]
output: [(None,)]
preprocessing: KerasLayer
input: None
output: 'input_type_ids': (None, 128), 'input_word_ids': (None, 128), 'input_mask': (None, 128)
BERT_encoder: KerasLayer
input: 'input_type_ids': (None, 128), 'input_word_ids': (None, 128), 'input_mask': (None, 128)
output: 'sequence_output': (None, 128, 768), 'pooled_output': (None, 768), 'encoder_outputs': [(None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768), (None, 128, 768)], 'default': (None, 768)
dense_3: Dense
input: (None, 768)
output: (None, 768)
dropout_41: Dropout
input: (None, 768)
output: (None, 768)
classifier: Dense
input: (None, 768)
output: (None, 5)

Figure 3: Ask Ubuntu Baseline

and "binary crossentropy" loss function for chatbot, as for the optimizer, we used the Adam optimizer with the default learning rate of 0.01 for all the models. We trained all the models on 5 epochs and kept the default batch size of 32 (Minibatch Gradient Descent).

We evaluated the accuracy and loss of each of the baseline models on the training and testing sets. The baseline scores are shown in Table 4. It is clear that each of the baseline models perform poorly on the corresponding data sets. In the next section we describe how we applied some fine-tuning to the models to improve these scores.

Equations to calculate the metrics are shown in (1-5). Where TP consists of the true positives, the sentences correctly classified as belonging to the correct intent, TN the true negatives, the sentences correctly classified as not belonging to the current intent, FP the false positives, the sentences predicted as belonging to the current intent but actually do not and FN the false negatives, the sentences predicted as not belonging to the current intent but actually do. For tasks with more than 2 instances, the accuracy is the weighted average of accuracies on all classes.

$$Sensisivity(TPR) = TP/(TP + FN) \tag{1}$$

$$Specificity(TNR) = TN/(TN + FP) \tag{2}$$

$$Precision = TP/(TP + FP) \tag{3}$$

$$F - 1 = 2 \times \frac{Precision \times Sensitivity}{Precision + Sensitivity} \tag{4}$$

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) \tag{5}$$

Data set	train_loss	train_acc	test_acc	test_loss
AskUbuntu	1.7413	0.1905	0.3394	1.4509
WebApps	1.9622	0.2917	0.1695	2.0031
Chatbot	0.6969	0.5	0.5	0.6969

Table 4: Baseline scores

5 Fine-tuning Models

5.1 1^{st} fine-tuning

In order to get the best hyper-parameters, the authors of [5] tried several combination of hyper-parameter values, which is computationally expensive and time consuming, therefore we tried another, simple fine-tuning, approach to get the best results on each data sets. This approach is based on [4].

For fine-tuning, we first calculated the maximum sequence length as defined in the Section 3, it equals to 33, 28 and 31 for askubuntu, webapps and chatbot, respectively. After setting the maximum sequence length and the 768 features, the lambda layer is used to convert the vectors dimensions from the BERT model layer to our desired dimensions before dropping out with a frequency of 0.5. We then inserted a dense layer and a dropout layer with the same frequency before doing the classifications with the final dense layer.

As for the optimizer, we also used the Adam optimizer but this time with a smaller learning rate of 0.00001. We used the same loss function for all the models which is Sparse categorical crossentropy. We used a stochastic gradient descent approach,i.e., batch size of 1, on the chatbot and webapps models and a minibatch gradient descent approach with a batch size of 3 for the askubuntu model. And we set the number of training epochs to 5.

The architecture of the fine-tuned askubuntu model is shown in Figure 4, the webapps and chatbot models also follow the same structure only that the number of outputs for the final dense layers are set to 8 and 2 respectively, representing the number of classes.

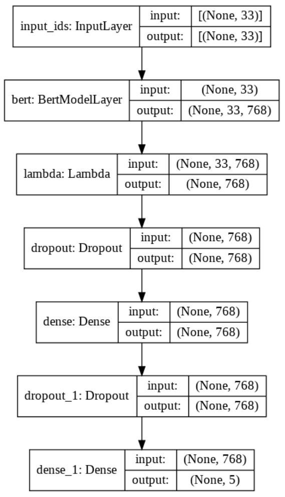


Figure 4: Fine-tuned Ask Ubuntu model

We trained the models until there was no improvement on the scores and in Table 5, we see the performance of each of them on the test and train sets. Even though the webapps model still performs poorly on the test set, when compared to the baseline model, we notice some improvement on all the scores. We notice a high accuracy of

98.11% and relatively lower loss of 0.0491 on the chatbot set, and a better accuracy of 89.91% on the askubuntu set as compared to the baseline model. This shows that there is still room for improvement, especially for the webapps model.

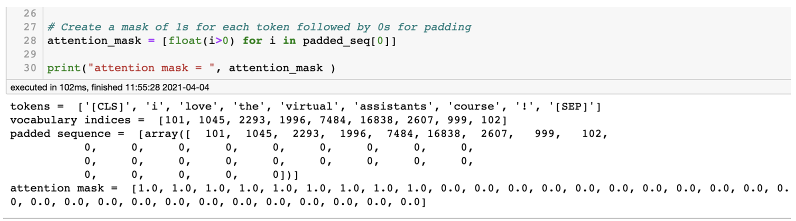
Data set	train_loss	train_acc	test_acc	test_loss
AskUbuntu	0.1945	0.9623	0.8991	0.4651
WebApps	0.3389	0.9333	0.4746	1.6632
Chatbot	0.0692	0.9900	0.9811	0.0491

Table 5: Scores after 1st fine-tuning

5.2 2nd fine-tuning

As in the first step of fine-tuning we achieved relatively good results on the chatbot and askubuntu data sets, in this step we will fine-tune the model to achieve better results on the webapps data set. As we can see in Table 3, we only have 24 samples for training, 6 for validation and 59 for testing. This clearly shows that we do not have enough training data as it is one of the requirements for proper generalization. In this step, we manually added one feature, which is "attention masks" in addition to the padded sequence of vocabulary indices used in the previous sections, and fed it to the BERT model. We also trained the model with pytorch in our defined training loop. For this task, we followed the steps in [10].

The attention mask is a sequence of 1s and 0s, with 1s for all input tokens and 0s for all padding tokens. For our previous example, i.e., "I love the Virtual Assistant Course", still with a maximum sequence length of 32, the attention mask is shown in Fig. 5, also with a length of 32. For the webapps data set, we kept the same maximum sequence length obtained in the previous section, i.e., 28.



```

# Create a mask of 1s for each token followed by 0s for padding
attention_mask = [float(i>0) for i in padded_seq[0]]

print("attention mask = ", attention_mask )
```

executed in 102ms, finished 11:55:28 2021-04-04

```
tokens =  ['[CLS]', 'i', 'love', 'the', 'virtual', 'assistants', 'course', '!', '[SEP]']
vocabulary indices =  [101, 1045, 2293, 1996, 7484, 16838, 2607, 999, 102]
padded sequence =  [array([  101,  1045,  2293,  1996,  7484, 16838,  2607,   999,   102,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0])]
attention mask =  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Figure 5: Attention Masks

For this task, we loaded the BertForSequenceClassification[1] model, which is the BERT model with one layer added on top of it for doing the classification from the 768 features from the default BERT model. Therefore, the additional layer consists of untrained linear neurons of size [hidden_state, number_of_labels], which is [768,8], i.e.,

[1] https://github.com/huggingface/pytorch-pretrained-BERT/blob/master/pytorch_pretrained_bert/modeling.py#L1129

the output of BERT plus our classification layer, a vector of 8 numbers representing the "score" for our labels that are then fed into cross-entropy loss.

We used the BertAdam optimizer, an optimizer that has been adapted to be closer to the optimizer used in the TensorFlow implementation of Bert; BertAdam differs from Adam in a sense that it does not compensate for bias as in Adam and it also implements weight decay fix. Therefore we grouped all the hyperparameters in BertAdam and set them with values of 0.01 and 0.0, with a learning rate of 0.00002. We also used an SGD approach (batch size of 1), and trainied over 5 epochs.

In our training loop, we took the following steps:

1. Set the model in train mode in order to compute the gradient.
2. After unpacking the data, clear the previously calculated gradients, and feed the unpacked data through the neural network.
3. Backpropagate
4. Update the parameters
5. Monitor the variables and scores

The evaluation loop also follows the same process, only that when evaluating, the model is set to "evaluation mode", i.e., it does not calculate and update the gradients, therefore it only calculates the loss on the validation set.

We tested with different levels data distribution of training and validation data and the best results were achieved with 10% validation data, i.e., 90% of training data of the whole training data set. The results after training with these levels are shown in Table 6.

Data set	train_loss	val_acc	test_acc	test_f-1
WebApps	0.8370	0.3333	0.6779	0.5968

Table 6: Scores after 2nd fine-tuning

Although the validation accuracy is low, 33.33%, we ignore that fact and move to testing because the validation set is not representative and only contains 10% of the whole training set (3 samples). As seen in Table 6, the test accuracy on this data set using this approach is 67.79% and is 20% higher than the one obtained with the previous approach, i.e., 47.46%. A quick look at the confusion matrix, in Table 7, in bold are the correctly classified samples (TPs), we can see the model performs best at classifying instances with the *Delete Account* and *Filter Spam* intent. These results are discussed in the next section.

6 Analysis and Discussion

As mentioned in the previous sections, the data sets are not representative enough for good model inference, therefore the baseline methods yielded poor results. In the 1st fine-tuning step, we have seen good accuracy improvements on the askubuntu, from 33.94% to 89.91%, and chatbot, from 50% to 98.11%, test sets as compared to the baseline methods. This shows that properly adding layers after extracting

	Chg_Pwd	Del_Acc	xprt_Data	Filter	Alt	None	Sync
Chg_Pwd	**2**	2	0	2	0	0	0
Del_Acc	0	**10**	0	0	0	0	0
xprt_Data	0	2	**0**	1	0	0	0
Filter	0	0	0	**14**	0	0	0
Alt	0	1	0	1	**14**	0	0
None	0	2	0	2	0	**0**	0
Sync	3	2	0	1	0	0	**0**

Table 7: Confusion matrix on test set

the embeddings with BERT from the input sentences can lead to improvements in accuracy. As also demonstrated in [5].

With the webapps data set on the other hand, we have more classes, 8 in total, but not enough data for proper generalization, therefore the baseline and 1^s fine-tuned models for webapps both performed poorly. By adding attention masks and decreasing the validation set size,i.e., increasing the train set size, in the 2^{nd} fine-tuning for webapps, we noticed a jump in accuracy as compared to the 1^s fine-tuned model, from 47.46% to 67.79%. This shows that adding attention masks as inputs for the BERT model can lead to accuracy improvements even when the data sets are relatively small.

7 Conclusion and Future Works

In this project, we have used Deep Learning and BERT (Bidirectional Encoder Representation for Transformers) for intent classification on three data sets, chatbot, webapps and askubuntu in an attempt to implement the authors' idea in [5]. Although these data sets are relatively small, the use of Deep Learning, especially BERT embeddings as features, has shown to be useful in intent classification. The BERT embeddings helped achieve relatively good results on the three data sets, reaching a 98.11% accuracy on the chatbot data set and 89.91% accuracy on the askubuntu data set without attention masks and a 67.67% accuracy on the smaller webapps data set with attention masks, the code and data sets have been uploaded on GitHub [6].

For future works, we plan using FastText embeddings [1] and compare the performance with BERT embeddings, and get more experience with Hyperas for fast hyper-parameters tuning to properly implement the idea in [5]. We also plan adding attention masks as input for the askubuntu and chatbot BERT models and observe if it will lead to improvements.

Acknowledgments

Special thanks go to Professor Caroline Barriere for her exemplary guidance and monitoring throughout the course of this project.

References

[1] Kaspars Balodis and Daiga Deksne. "FastText-Based Intent Detection for Inflected Languages". en. In: *Information* 10.5 (May 2019). Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, p. 161. DOI: 10.3390/info10050161. URL: https://www.mdpi.com/2078-2489/10/5/161.

[2] Daniel Braun et al. "Evaluating Natural Language Understanding Services for Conversational Question Answering Systems". In: *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*. Saarbrücken, Germany: Association for Computational Linguistics, Aug. 2017, pp. 174–185. DOI: 10.18653/v1/W17-5522. URL: https://www.aclweb.org/anthology/W17-5522.

[3] McCormick C. *BERT Word Embeddings Tutorial*. URL: https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/#2-input-formatting.

[4] *Intent Recognition with BERT using Keras and TensorFlow 2*. en-US. 2020. URL: https://www.kdnuggets.com/intent-recognition-with-bert-using-keras-and-tensorflow-2.html/.

[5] Jurgita Kapočiūtė-Dzikienė, Kaspars Balodis, and Raivis Skadiņš. "Intent Detection Problem Solving via Automatic DNN Hyperparameter Optimization". en. In: *Applied Sciences* 10.21 (Jan. 2020). Number: 21 Publisher: Multidisciplinary Digital Publishing Institute, p. 7426. DOI: 10.3390/app10217426. URL: https://www.mdpi.com/2076-3417/10/21/7426.

[6] Paul Mvula. *womega/CSI5180_Project*. original-date: 2021-03-15T04:57:17Z. Apr. 2021. URL: https://github.com/womega/CSI5180_Project.

[7] Max Pumperla. *maxpumperla/hyperas*. original-date: 2016-02-19T14:45:10Z. Mar. 2021. URL: https://github.com/maxpumperla/hyperas.

[8] Reza Rawassizadeh et al. "Manifestation of virtual assistants and robots into daily life: vision and challenges". en. In: *CCF Transactions on Pervasive Computing and Interaction* 1.3 (Nov. 2019), pp. 163–174. ISSN: 2524-5228. DOI: 10.1007/s42486-019-00014-1. URL: https://doi.org/10.1007/s42486-019-00014-1.

[9] Mike Schuster and Kaisuke Nakajima. "Japanese and Korean voice search". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Kyoto, Japan: IEEE, Mar. 2012, pp. 5149–5152. ISBN: 978-1-4673-0046-9 978-1-4673-0045-2 978-1-4673-0044-5. DOI: 10.1109/ICASSP.2012.6289079. URL: http://ieeexplore.ieee.org/document/6289079/.

[10] *Training a Classifier — PyTorch Tutorials 1.8.1+cu102 documentation*. URL: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py.

[11] Ashish Vaswani et al. “Attention Is All You Need”. In: *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.