WARSAW UNIVERSITY OF TECHNOLOGY
FACULTY OF MATHEMATICS AND INFORMATION SCIENCE

PROJECT 2: REPORT

Speech commands classification with recurrent neural networks

Dora Doljanin, Filip Pavičić

Warsaw, May 2022.

Table of contents

Introduction

As digital technologies, gadgets, and machines continue to play a significant role in our everyday lives, a need to improve human and machine interaction has become inevitable. The problem of speech command recognition has gained a lot of interest in the last years, and it continues to advance. A wide number of industries are utilizing different applications of speech technology today, helping businesses and consumers save time and even lives. Some examples include automotive, virtual agents, health care, sales, security, etc. [4]

The problem of speech command recognition can be tackled using various algorithms and models, including architectures based on recurrent neural networks and long short-term memory (LSTM). This project aims to solve the problem of classifying speech commands using deep learning and recurrent neural networks. The goal is to test and compare different network architectures.

In the following chapter, we will first describe the research problem. Then, in the second chapter, we will provide a theoretical introduction to recurrent neural networks, LSTM, and speech commands classification. In the third chapter, we will present the application developed to solve the proposed problem and provide an overview of the technologies and tools used in product implementation. In the last two chapters, we will explain the conducted experiments and analyse the results.

1. Research problem

1.1. Description of the problem

We aim to solve the problem of speech commands classification, which is the task of classifying an input audio pattern into a discrete set of classes. [3] We use the Speech Commands Data Set dataset, version 0.01 [2]. It is a set of 64,727 one-second .wav audio files, each containing a single spoken English word. These words are from a small set of commands and are spoken by a variety of different speakers. The audio files are organized into folders based on the word they contain, and this data set is designed to help train simple machine learning models. The core words are "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", and "Nine". To help distinguish unrecognized words, there are also ten auxiliary words, which most speakers only said once. These include "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree", and "Wow". We want to solve the proposed problem using deep learning and recurrent neural networks. We further want to test and compare the performances of different neural network architectures by conducting several experiments.

1.2. Requirements

The requirements are the following:

- At least one of the used network architectures should be Long short-term memory (LSTM)
- We want to investigate influence of parameters change on the obtained results
- We shall present a confusion matrix of the obtained results
- We should pay special attention on “silence” and “unknown” classes and test different approaches (e.g., separate network for their recognition)

2. Theoretical introduction

2.1. Speech command classification

Speech recognition technologies are present in several areas of our lives, including widely used personal assistants like Google Assistant, Microsoft Cortana, Amazon Alexa, and Apple Siri. [6] It also includes self-driving car human-computer interactions and activities where employees need to wear lots of protection equipment (like the oil and gas industry). [6] However, classifying human speech is a challenging task. It is considered one of the most complex areas of computer science since it involves linguistics, mathematics, and statistics. [4] Several factors can impact the performance of speech recognition models, such as pronunciation, accent, pitch, volume, and background noise. [4] One of the most successful approaches to solving the speech commands classification task is using deep learning and recurrent neural networks. This has been demonstrated by several models which have shown high-quality performances. [6][7][8][9]

2.2. Recurrent neural networks

A recurrent neural network (RNN), just like other neural networks, has a node that receives an input, processes it, and results in an output. However, what makes an RNN node different from other neural networks is the fact that it is recurrent, meaning that it loops around. This means that the output of the given step is provided alongside the input of the next step. This allows the network to remember previous steps in a sequence and to use it as contextual information when mapping between input and output. [1]

However, RNN suffers from the long-term dependency problem: over time, as more and more information piles up, the RNN becomes less effective at learning new things. [1] The problem is that the influence of a given input on the network output either decays or blows up exponentially as it cycles around the network's recurrent connections. [5] This effect is called the vanishing gradient problem. This is where Long short-term memory (LSTM) comes into play. LSTM is a type of recurrent neural network that is able to remember information it needs to keep hold of the context, but also to forget information that is no longer applicable. [1]

2.3. LSTM

LSTM networks have feedback connections, unlike more traditional feedforward neural networks. This enables them to process entire sequences of data without treating each point in the sequence independently. They rather retain useful information about previous data in the sequence to help them process new data. As a result, LSTMs are particularly successful in processing sequences of data such as text, speech, and general time series. [10] Long short-term memory provides a solution for the long-term dependency problem by adding the internal state to the RNN node. The internal state is a cell that consists of three parts: forget gate, input gate, and output gate. [1] The forget gate determines what kind of state information is no longer relevant to the context and can therefore be forgotten. The input gate says what new information should be added to the state information. The output gate chooses which part of it should be output out of all the information stored in the state for a particular instance. Each gate is assigned a number between 0 and 1. The number 0 means that the gate is closed, and nothing gets through, while 1 means that the gate is wide open and everything gets through. That way, we can control how much information gets forgotten, input or output. LSTM is typically used for machine translation, Q&A chatbots etc. [1]

3. Instruction of the application

We implemented the application in the Python programming language in the form of a Jupyter notebook. We used the PyTorch framework and the TensorFlow platform. In addition to that, we utilized the resources provided by Google Collab and Kaggle. The following are the details of product implementation and the corresponding code snippets.

3.1. Code

3.1.1. Preparing the dataset

We downloaded the dataset from Kaggle into the working environment's file system and unzipped the test and train dataset. We split the original training dataset on training and validation datasets.

For loading the data, we created our class *SpeechDataset* which has three getters for retrieving train set, validation set and train + validation set. The *SpeechDataset* class uses a private class *_SpeechSubset* that inherits PyTorch's *Dataset* class.

```
class SpeechDataset():
    def __init__(self, root, train_size = None, exclude = None, transformation = None, ):
        self.root = root
        self.train_size = train_size
        self.transformation = transformation
        self.dataset = _load_folder_list(self.root)

        if exclude is not None:
            self.dataset = [x for x in self.dataset if exclude(x) == False]

        if self.train_size is not None:
            self.train_subset, self.val_subset = train_test_split(self.dataset, train_size=self.train_size)

    def __getSubset(self, name):
        if name == "train":
            return _SpeechSubset(self.root, self.train_subset, transformation = self.transformation)
        elif name == "val":
            return _SpeechSubset(self.root, self.val_subset, transformation = self.transformation)
        elif name == "train+val":
            return _SpeechSubset(self.root,  self.dataset, transformation = self.transformation)
        else:
            return None

    def getTrainSubset(self,):
        return self.__getSubset("train")

    def getValidationSubset(self):
        return self.__getSubset("val")

    def getTrainValidationSubset(self):
        return self.__getSubset("train+val")

    def getFilenames(self):
        return self.dataset
```

1. *SpeechDataset* class for loading the data

```
class _SpeechSubset(Dataset):
    def __init__(self, root, subset, transformation = None):
        self.root = root
        self.subset = subset
        self.transformation = transformation

    def __len__(self):
        return len(self.subset)

    def __getitem__(self, idx):
        filepath = self.subset[idx]
        label, waveform, sample_rate = load_speechcommands_item(self.root, filepath)
        if self.transformation is not None:
            label, waveform, sample_rate = self.transformation(label, waveform, sample_rate)
        return waveform, sample_rate, label,
```

2. Private class *_SpeechSubset*

After that, we split the data into training and validation sets, where the validation set contains 30% of the original labelled data. Furthermore, for each dataset we created an instance of the *DataLoader* class which will provide batches for further training.

3.1.2. Creating the “silence” class

In the original dataset there are only six voice recordings classified as “silence”. For each of them, we created 6 different recordings by changing the volume of sound from minimum value (approximately 0% of original volume), to 20%, 40%, 60%, 80%, and finally 100% of the original volume. After that, we chopped each of the recordings to clips with the duration of 1 seconds. The result is a set of 2412 examples of the “silence” class.

```
shutil.rmtree('./train/audio/silence', ignore_errors=True)
Path("./train/audio/silence").mkdir(parents=True, exist_ok=True)

low_sum_by_second = 0.5

max_ratio_level = 1
multiplication = 5


for z,(w,s,l) in enumerate(whole_set_noise):
    min_val = len(w[0]) / (s * float(w[0].abs().sum())) * low_sum_by_second
    step = (max_ratio_level - min_val) / multiplication
    print()
    for i in range(multiplication +1):
        ratio = min_val + step * i
        wave = w * ratio
        wave_split = wave.split(16000, dim = 1)
        for j,w1 in enumerate(wave_split):
            Path("./train/audio/silence").mkdir(parents=True, exist_ok=True)
            torchaudio.save('./train/audio/silence/'+str(z) + "_" +str(i) + "_" + str(j)+'.wav', w1, 16000)
```

3. Creating the "silence" class

3.1.3. Defining the models

We used two models which have the same structure of layers, but handle classification differently. The first model trains on examples labelled as original classes from the dataset + the class “silence” (31 classes total) and later transforms every useless class into the class “unknown” (12 classes total). On the other hand, the second model firstly transforms every useless class into the class “unknown” and then trains on these classes (12 classes total).

```
class RNN2(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN2, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.conv1 = nn.Conv1d(1, 32, kernel_size=80, stride=4)
        self.bn1 = nn.BatchNorm1d(32)
        self.pool1 = nn.MaxPool1d(4)
        self.conv2 = nn.Conv1d(32, 64, kernel_size=16, stride=1)
        self.bn2 = nn.BatchNorm1d(64)
        self.pool2 = nn.MaxPool1d(2)
        self.lstm = nn.LSTM(980, hidden_size, num_layers, batch_first=True, dropout = 0.3)
        self.fc1 = nn.Linear(hidden_size, 128)
        self.fc = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = self.pool1(x)
        x = F.dropout(x, 0.2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        #x= self.pool2(x)
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).cuda()
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).cuda()

        # Forward propagate LSTM
        out, (h,c) = self.lstm(x, (h0, c0))  # out: tensor of shape (batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = out[:, -1, :]
        out = self.fc1(out)
        out = F.dropout(out, 0.3)
        out = self.fc(out)
        return out
```

4. The structure of both models

```
model4 = RNN2(16000, 512, 4, len(labels)).cuda()
```

5. Initialization of a model

The structure of layers is the following: we used two convolutional layers, after that one LSTM layer which has 4 internal layers, and finally 2 fully connected layers.

Model 1

The first model is more complex than the second model because it works with 31 classes while training. However, it escapes the problem of imbalanced classes, unlike the second model.

Model 2

After transforming every useless class into the class “unknown”, the class “unknown” had much more instances than any other class. This is known as the problem of imbalanced classes. To deal with the problem, we used penalized classification. We added weights in the cross-entropy loss function which impose additional cost for making classification mistake on the minority class during training. That way, the model pays more attention to the minority class, and less on the majority class (“unknown”). [11]

3.1.4. Training

We created a training method that uses the *Adam* optimizer, which executes the stochastic gradient descent algorithm. In addition to that, we decided to decrease the learning rate to 10% of the previous value every 25 epochs.

```
def train(model, train_dataloader, test_dataloader = None, num_epochs=1, lr=1e-3, keepBest = False, metrics = None,
          weight = None, augumentation = None):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.1)

    acc_train = np.empty([0])
    acc_val = np.empty([0])

    best_model_wts = None
    best_acc = None

    for epoch in range(num_epochs):
        correct_train = 0
        model.train()

        with tqdm(enumerate(train_dataloader), total=len(train_dataloader), desc=f'Training (epoch={epoch}/{num_epochs})')
        as epoch_progress:
            for batch_idx, train_batch in epoch_progress:
                x, y = train_batch
                x = x.cuda()
                y = y.cuda()
                if augumentation is not None:
                    x = augumentation(x)
                logits = model(x)

                loss = F.cross_entropy(logits.squeeze(), y, weight = weight)
                logits_argmax = logits.argmax(dim = -1).squeeze()
                epoch_correct = (logits_argmax==y).sum().item()


                correct_train += epoch_correct
                epoch_progress.set_postfix({'batch acc': epoch_correct / x.shape[0]})

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
```

6. The training method - part 1

```
        train_cc = correct_train / len(train_dataloader.dataset)
        acc_train = np.append(acc_train,train_cc)
        print("Train acc " + str(train_cc))
        scheduler.step()

        if test_dataloader is not None:
            with torch.no_grad():
                model.eval()
                correct_val = 0
                for test_batch in tqdm(test_dataloader, desc="Testing"):
                    x, y = test_batch
                    x = x.cuda()
                    y = y.cuda()

                    logits = model(x)

                    logits_argmax = logits.argmax(dim = -1).squeeze()
                    epoch_correct = (logits_argmax==y).sum().item()

                    if metrics is not None:
                        metrics.insert(logits_argmax, y)

                    correct_val += epoch_correct

                val_acc = correct_val / len(test_dataloader.dataset)
                acc_val = np.append(acc_val,val_acc)
                print("Val acc " + str(val_acc))

                if metrics is not None:
                    metrics.print()

                if best_acc is None or best_acc < val_acc:
                    best_acc = val_acc
                    best_model_wts = copy.deepcopy(model.state_dict())

    if keepBest and test_dataloader is not None:
        model.load_state_dict(best_model_wts)

    model.train()
    return acc_train, acc_val
```

7. The training method - part 2

3.1.5. Defining the hyper-parameters

The hyper-parameters we used are the following:

Regularization hyper-parameters

- ***Dropout rate*** – the percentage of nodes that will be turned off in certain layers, which helps the networks to widen the variety of nodes that will be activated in the training of each example

Training hyper-parameters

- ***Batch size*** – the number of examples in one training group
- ***Learning rate*** – the number that multiplies the gradient, which determines how much the weights will be adjusted in the direction opposite of the gradient
- ***Shuffle*** – a Boolean value which determines whether we will mix the examples between the batches in each epoch
- ***Number of epochs*** – this parameter will be determined according to the results on the validation set (when the validation loss reaches its minimum value)

4. Description of the conducted experiments

4.1. Examining a difference between validation accuracy and Kaggle test data accuracy

We managed to obtain more than 90% accuracy on validation dataset for both models, so it was surprising to see that on Kaggle test data we obtained approximately 78% accuracy. Since that is a significant difference, we wanted to investigate the reasoning behind the observed behaviour. We decided to find the examples in the Kaggle test dataset whose classes our models were not very confident about. We recognised them by applying the *SoftMax* function on the models' outputs and sorting examples by the second highest value of the SoftMax function. The results will be explained in the following chapter.

4.2. Examining an influence of parameters' change on the obtained results

To investigate the influence of the hyper-parameters on obtained results, we trained the model on the training dataset for certain combinations of hyper-parameters (e.g., dropout rate, learning rate, batch size, etc.) and validated them on the validation dataset. The optimal hyper-parameters are those that minimise the value of the loss function on the validation set. We conducted several experiments by changing the hyper-parameters, training the model, and observing the effect on the validation loss. In addition to that, we tried to narrow the gap between the training loss and validation loss, because a high value of the gap can indicate a possibility of over-fitting. The results will be explained later.

5. Result analysis

5.1. Results

While examining the Kaggle test dataset, we found that it has a lot of noise. We also found that there are some examples in the Kaggle test dataset that do not belong to any of the original classes, so they should be labelled as unknown. We assume that our model is confused with that sort of data, which might be the reason why we have that accuracy gap between the validation and Kaggle test dataset.

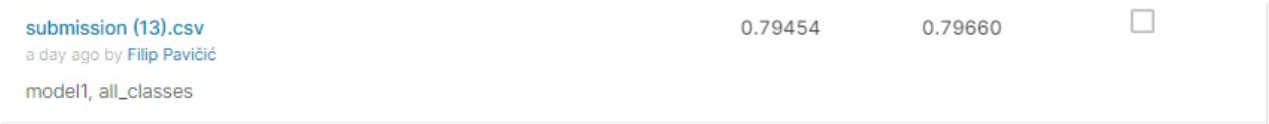
5.1.1. Model 1

After 30 epochs, we managed to achieve an accuracy of 94.21%. Additionally, we tried training the model with different sets of hyperparameters. However, we did not achieve any significant changes of the accuracy.

```
Training (epoch=29/30): 100%|██████████| 184/184 [00:39<00:00,  4.71it/s, batch
acc=0.972]
Train acc 0.9803374970740323
Testing: 100%|██████████| 79/79 [00:10<00:00,  7.62it/s]
Val acc 0.9088877855014895
UnkownPerformanceMetric:  0.9421052631578948
```

8. Train and validation accuracy of the first model

After having trained the model on the whole dataset (train + validation), we managed to achieve an accuracy of 79.66% on the test dataset.



9. Accuracy of the first model on the test dataset

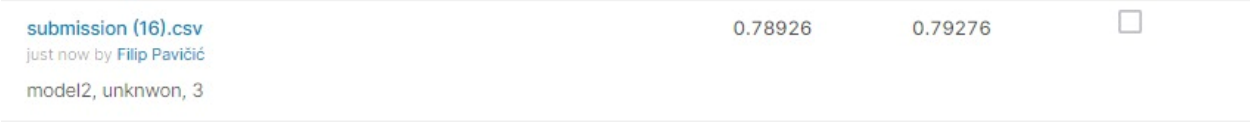
5.1.2. Model 2

After 30 epochs, we managed to achieve an accuracy of 92.34%. Additionally, we tried training the model with different sets of hyperparameters. However, we did not achieve any significant changes of the accuracy.

```
Training (epoch=29/30): 100%|██████████| 184/184 [00:39<00:00,  4.72it/s, batch
acc=0.993]
Train acc 0.9695060966526929
Testing: 100%|██████████| 79/79 [00:10<00:00,  7.77it/s]
Val acc 0.9234359483614697
NotUnkownPerformance:  0.8956665395857161
```

10. Train and validation accuracy

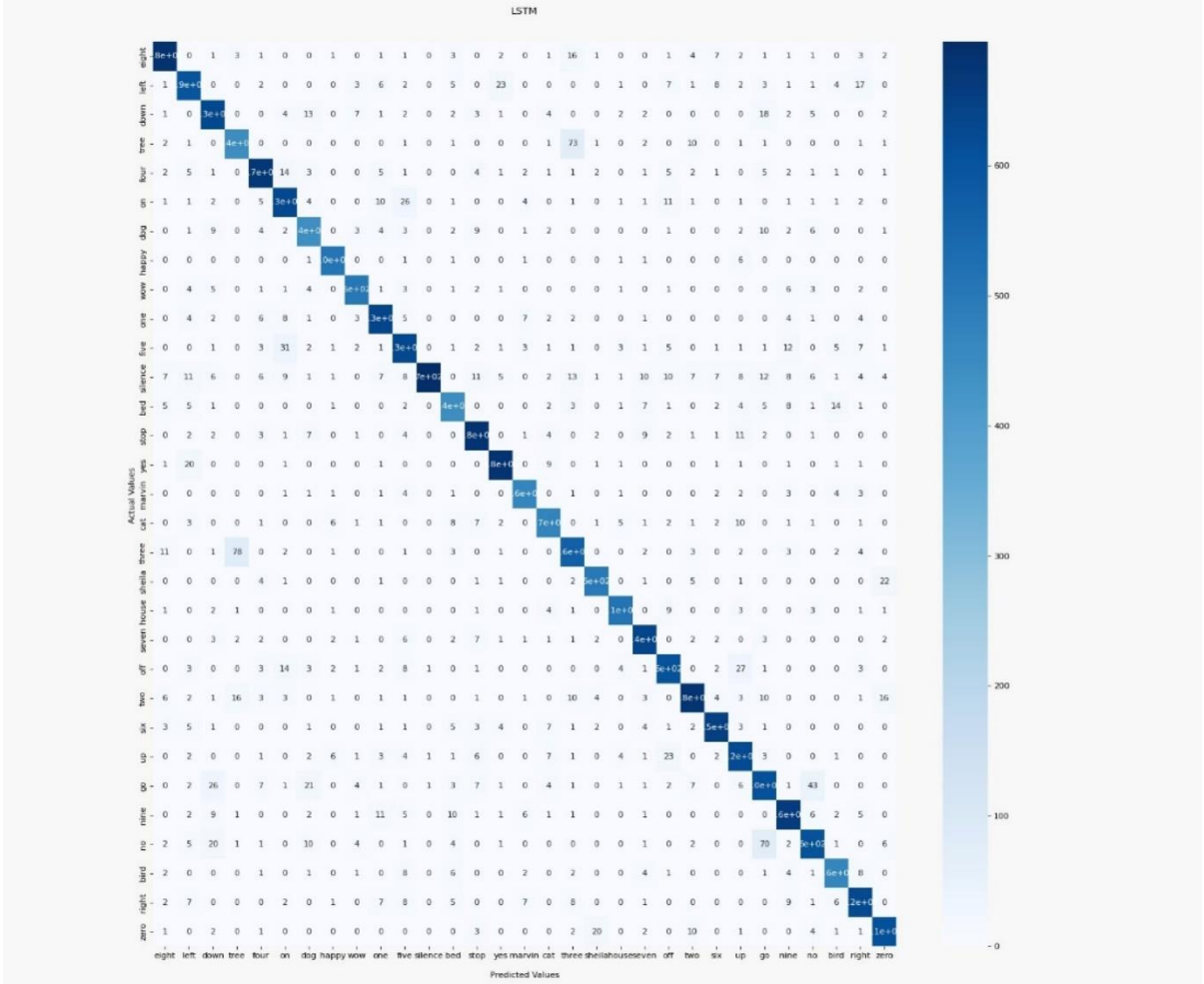
We trained this model on the whole dataset (train + validation) and obtained an accuracy of 79.28% on the test dataset.



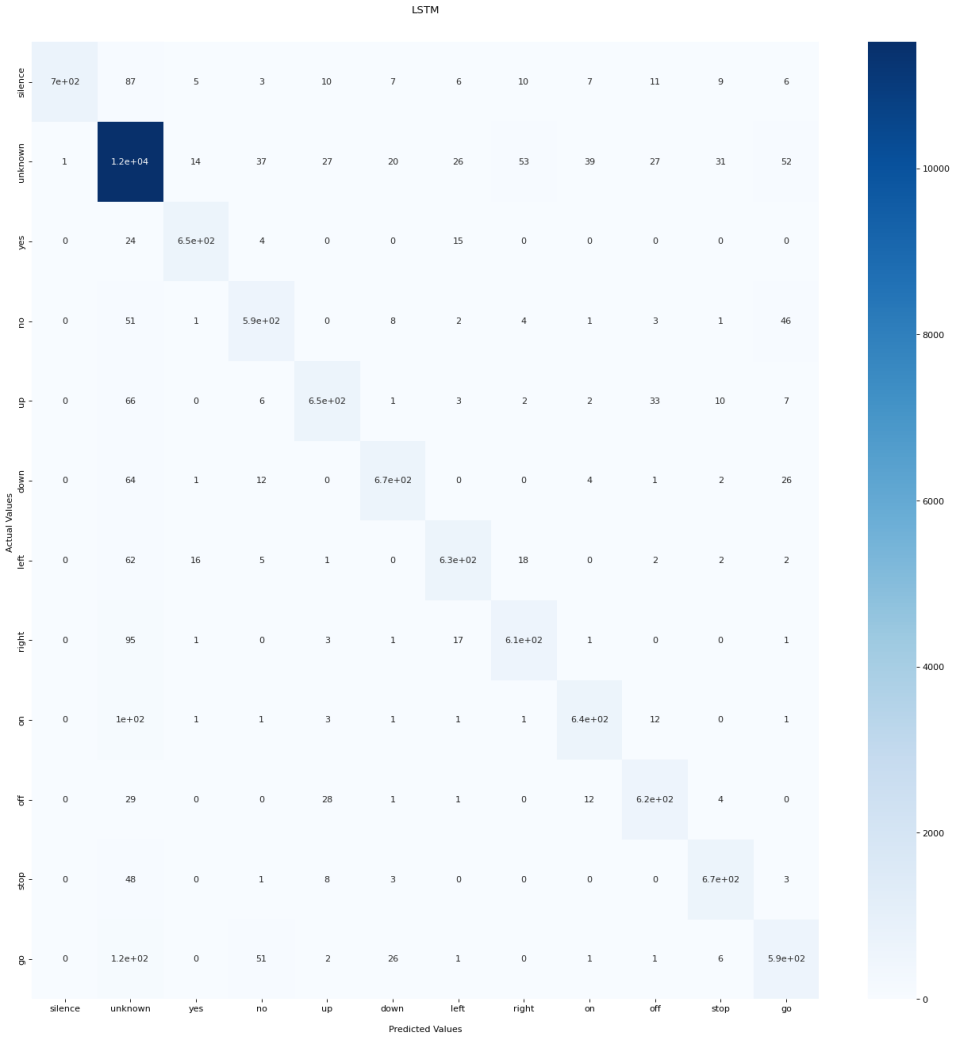
11. Accuracy of the second model on the test dataset

5.2. Comparison

The first model achieved the accuracy of 79.66%, while the second model achieved the accuracy of 79.28% on test dataset. From the results obtained, we may conclude that the two models obtained very similar performances. This was surprising for us because we expected that one of the models would outperform the other more significantly. To further observe the performances and behaviour of both models, we created corresponding confusion matrices of the models on the validation sets.



12. Confusion matrix of the first model on validation dataset



13. Confusion matrix of the second model on validation dataset

From the given matrices, we may draw useful conclusions about the models. We can see that the first model has some problems with distinguishing “Three” and “Tree”, as many trees have been classified as threes, and vice versa. This is not surprising, because these two words sound very similar and even many humans may mishear them in everyday conversations. Similarly, we may notice that the second model has some problems with distinguishing “No” and “Go”, for the same reason. Furthermore, we may notice that the second model makes the most mistakes with the class “Unknown”. This is an expected behaviour, as the classes are imbalanced, and the most examples belong to this class. For this reason, we must be careful when drawing conclusions from the second confusion matrix. It might be better to observe percentages, rather than quantities, for more accurate conclusions.

Conclusion

In this project, we aimed to solve the problem of classifying speech commands using deep learning and recurrent neural networks. We implemented, tested, and compared different network architectures to find the best possible solution. While tackling the problem of speech commands classification, we faced many challenges, including unclear pronunciation, unusual accent, unexpected pitch, different variations of volume, and background noise. All of that assured us the complexity of the proposed problem.

However, we managed to achieve the accuracy slightly above 79% on both models. According to the obtained results, we may conclude that we have successfully solved the problem of speech commands classification. As a follow up to this project, the next challenge could be to create an attention-based model for speech recognition which iteratively processes the input by selecting relevant content at every step. [12]

Literature

[1] IBM Technology, *What is LSTM (Long Short Term Memory)?*, Link: https://www.youtube.com/watch?v=b61DPVFX03I

[2] TensorFlow Speech Recognition Challenge, Link: https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data

[3] Nvidia Docs, *Speech Classification*, Link: https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/stable/asr/speech_classification/intro.html

[4] IBM, Speech recognition algorithms, Link: https://www.ibm.com/cloud/learn/speech-recognition

[5] Graves, Alex. "Long short-term memory." *Supervised sequence labelling with recurrent neural networks*. Springer, Berlin, Heidelberg, 2012. 37-45.

[6] de Andrade, D. Coimbra. "Recognizing Speech Commands Using Recurrent Neural Networks with Attention." (2018).

[7] Sak, Haşim, et al. "Fast and accurate recurrent neural network acoustic models for speech recognition." *arXiv preprint arXiv:1507.06947* (2015).

[8] Graves, Alex, and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks." *International conference on machine learning*. PMLR, 2014.

[9] de Andrade, Douglas Coimbra, et al. "A neural attention model for speech command recognition." *arXiv preprint arXiv:1808.08929* (2018).

[10] Rian Dolphin, *LSTM Networks | A Detailed Explanation*, Link: https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9

[11] Jason Brownlee, “8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset”, August 19, 2015. Link: https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

[12] Chorowski, Jan K., et al. "Attention-based models for speech recognition." Advances in neural information processing systems 28 (2015).