University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Piotr Ambroszczyk
Student no. 385090

Łukasz Kondraciuk
Student no. 385775

Wojciech Przybyszewski
Student no. 386044

Jan Tabaszewski
Student no. 386319

Deep Learning for speech recognition – Deep Speech 2

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Instytut Informatyki

June 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Bachelor of Computer Science.

Date Supervisor's signature

Authors' statements

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Authors' signatures

Abstract

The authors of this thesis focus on implementing scripts for training DeepSpeech2 model for Automatic Speech Recognition. We try to reproduce results obtained by Baidu Research in End-to-End Speech Recognition paper [2] using `PyTorch` framework and techniques for speeding up computations on GPUs – training in mixed precision and mutli-GPU scaling.We provide fully trained models for English language together with statistics about how changing hyperparameters and architecture impacts model's performance and accuracy.

Keywords

Deep Speech, ASR, Neural Networks, Machine Learning, Python, PyTorch, NVIDIA, RNN, multi-GPU, FP16, AMP

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

D. Software

Tytuł pracy w języku polskim

Rozpoznawanie mowy metodami głębokiego uczenia – Deep Speech 2

Contents

Introduction

Transcription of spoken language is a crucial problem for many areas of a modern technology industry. Being able to communicate with electronic devices not only by touching them, but also by talking to them is an important goal for IT companies. Such devices are more user-friendly so it is undoubtedly beneficial for everybody. To achieve this goal, various solutions were proposed, and many of them use complex algorithms (e.g. Hidden Markov Models) [1]. However, it has been shown that the best accuracy can be achieved with Automatic Speech Recognition (ASR) models based on neural networks [2].

Our thesis concentrates on implementing state of the art ASR model Deep Speech 2 (DS2) described in [2] and we realize it with the support of NVIDIA Corporation. Authors of Deep Speech 2 showed that accuracy of the model depends not only on its implementation, but also on the size and diversity of used dataset (therefore, we have to find appropriate one and prepare it adequately). Large size of the dataset creates another problem – we need our model to be able to train on that data in a reasonable time. We also want it to work in real time after training. Last but not least, in order to determine the best hyperparameters we have to run many experiments, collect their results, and finally analyze them.

In order to accomplish our goals we are going to implement DeepSpeech2 model using `PyTorch` deep learning framework, which supports parallel processing using GPUs. To achieve high performance system, we plan to use open-source libraries prepared by NVIDIA, which make it possible to train one neural network on multiple GPUs. Another optimization, which we expect to speed up computations, is using half precision floating-point numbers (also known as FP16) together with single precision. When it comes to collecting datasets, we plan to use LibriSpeech dataset available online, so our results are easy to reproduce.

The structure of our thesis is as follows. In Chapter 1 we introduce architecture of Deep Speech 2 model in terms of, among others, used layers, data flow and functions. After that, in Chapter 2, we present applied optimizations which increased network performance together with experiments results. Finally, in Chapter 3, we summarize all our results and achieved goals.

We worked agile and therefore our work was divided between everyone. However, each person in our team had it main responsibilities. Jan mainly took care of audio encoding and transcription decoding. Łukasz was responsible for dataset preparation and porting model to mixed precision. Piotr took care if implementing neural network features and researching for the best model hyperparameters. Wojciech was responsible for implementing basic model and overall coordination of our work.

Chapter 1

Model description

DeepSpeech 2 (DS2) is a recurrent neural network trained to ingest speech spectrograms and generate a text transcription. Following description of the model architecture is quite basic and it is based on the original paper [2]. More details can be found there.

1.1. Input and Output specification

Let $\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), ...\right\}$ be a training set. $x^{(i)}$ is a time-series of variable length where every time-slice is a spectrogram of power normalized audio clips, so $x^{(i)}_{t,p}$ denotes the power of the p'th frequency bin in the audio frame at time t. $y^{(i)}$ is a transcription of the utterance $x^{(i)}$. One can see that $x^{(i)}$ is in fact a matrix of size depending on the number of frequency bins and i'th audio clip length. We use the same number of frequency bins for all audio clips in dataset.

DS2 network's input is a time-series x and the output is a prediction over characters $p(l_t|x)$ for each input time-step. For English language possible values of l_t are letters from 'a' to 'z', space, apostrophe and blank. Adding non-letter characters allows to find word boundaries. The special symbol blank allows repeating identical letters (e.g. "too") and generally helps generating correct transcriptions.

1.2. Layers

The model of the network is composed of one or more convolutional layers, followed in turn by one or more bidirectional recurrent layers [12], followed by one or more fully connected layers. Activation function used throughout the network is the Clipped Rectified Linear Unit (Clipped ReLU) function, given by the formula:

$$\sigma(x) = \begin{cases} 0 & \text{for } x < 0, \\ 20 & \text{for } x > 20, \\ x & \text{otherwise.} \end{cases}$$

The recurrent layers appear in a few different variants – standard recurrent layers or Long Short-Term Memory (LSTM) [13], or Gated Recurrent Units (GRU) [14]. After the recurrent layers and fully connected layers are applied, we count the output layer L as a softmax of the output of the last layer.

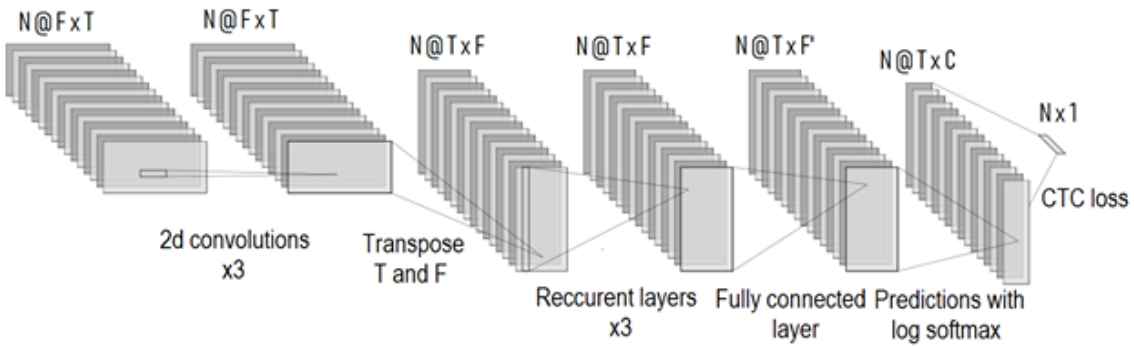

Figure 1.1: Model of the network. F - number of frequency bins, T - number of time frames, C - size of the alphabet

Softmax function $f : \mathbb{R}^k \to \mathbb{R}^k$ is defined by the formula:

$$f_i(v) = \frac{e^{v_i}}{\sum_{i=1}^{k} e^{v_j}},$$

and

$$f(v) = \big(f_1(v), f_2(v), \ldots, f_k(v)\big).$$

We basically apply exponential function to each outputted value, and then normalize these values to make sure, that probabilities sum up to 1. In our case $k = 29$, hence there is 29 possible output characters to distribute probability on (26 letters and 3 special symbols, as described in the previous section).

1.3. CTC Loss

To train a neural network, we typically need a function that would tell us how good current network's output is. The lesser value this function has, the better results our model achieves. This kind of function is called a **loss function**. Usually, minimizing a value of the loss function is a main goal of the training.

Loss function used in DS2 is **Connectionist Temporal Classification** (CTC) [3]. To define this loss, let us introduce an encoding of a text. Encoding of a given text S is done by replacing every character c in S by any number of characters c and blanks "-". Only restriction is that if there are two adjacent identical letters in S, they must be separated by a blank "-".

For instance, possible encodings of *"to"* are *"-tttooo"* and *"-tttoo-o"*, but only the latter could be an encoding of the word *"too"*.

Now we say that the probability of an actual transcription is the sum of probabilities of all possible encodings of the actual transcription, that have the same length as the output of the model. The loss function here is simply the negative logarithm of this probability. Having this we can count derivatives of this loss function with respect to model output and then apply *backpropagation through time* algorithm to train the network, which is just a standard backpropagation algorithm modified for recurrent neural networks.

1.4. Training

During the training, datasets are divided into batches of some certain size. Batches are fed one by one into the model and after each one CTC loss is computed and backpropagated through the network, and then weights on the layers are updated. The speed in which changes are applied to the model depends on some chosen parameter – **learning rate**. All batches make up for an epoch of training and training lasts for many epochs – until loss does not decrease anymore or decreases very slightly. Accuracy of the model is measured by **Word Error Rate** (WER). It is a common metric of the performance of speech recognition systems. Many commercial ASR systems are beeing compared using WER [5]. To measure Word Error Rate, we compare the recognized word sequence with a reference word sequence by transforming the latter to the former and find:

- S – the number of substitutions,
- D – the number of deletions,
- I – the number of insertions,
- C – the number of correct words,
- N – the number of words in the reference ($N = S + D + C$).

Now the WER can be computed as:

$$\text{WER} = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}.$$

1.5. Generating transcription

In order to find Word Error Rate of the model, one has to generate some final transcription. It is generated based on both output of the model and an N-gram language model [6]. To generate the final transcription, we search for transcription y that maximizes $Q(y)$, where Q is given as follows:

$$Q(y) = \log\left(\mathbb{P}_{ctc}(y|x)\right) + \alpha \cdot \log\left(\mathbb{P}_{lm}(y)\right) + \beta \cdot \text{word\_count}(y).$$

The term $\mathbb{P}_{ctc}(y|x)$ denotes the probability of y being a transcription of the utterance x as described in section 1.3 and the term $\mathbb{P}_{lm}(y)$ denotes the probability of the sequence y according to the language model. The weights α and β are tunable parameters and control the contribution of the language model and word count in scoring of transcription. We use a beam search similar to one described in [4] to find the optimal transcription y.

Beam search is an algorithm that iterates over each time-step in the output of DS2 model, while remembering some constant number (let this number be BW – *Beam Width*) of the most probable transcriptions up to this point. When considering the next time-step, we create $BW \cdot C$ new transcriptions corresponding to appending every possible character (we denote number of them by C) to the end of each of BW remembered transcriptions. After merging identical transcriptions and recalculating their probabilities we again keep BW best ones. In the end we have BW *most probable* transcriptions, so we just take the best one.

In order to efficiently merge transcriptions and recalculate new probabilities for them, we must store probabilities $\mathcal{P}_b$, $\mathcal{P}_{nb}$, $\mathcal{P}_{lm}$ for all kept transcriptions, where:

- $\mathcal{P}_b(y, t)$ – probability of the given transcription y at time-step t if the encoding ends with *blank*,
- $\mathcal{P}_{nb}(y, t)$ – probability of the given transcription y at time-step t if the encoding doesn't end with *blank*,
- $\mathcal{P}_{lm}(y)$ – probability of the given transcription y, according to the language model.

1.6. Summary

To sum up, workflow in DS2 is standard as for neural network. We prepare some data in the form described in the section 1.1. Next we forward propagate the data through the layers described in section 1.2. After that we count CTC Loss (section 1.3) and update model weights with specific learning rate (section 1.4). To evaluate the model we generate transcription from predictions given by the output layer using Beam search algorithm (section 1.5) and count Word Error Rate metrics (section 1.4).

Chapter 2

Experiments on architecture and hyperparameters

2.1. Mixed precision training

In order to speed up training, we decided to try to reduce precision of floating-point calculations, where full precision is not really needed to achieve comparable accuracy.

Standard deep learning frameworks, such as PyTorch, use 32 bit floating points variables (FP32) to keep model parameters, and perform all operation. Massively parallel architecture of modern GPUs allows to, perform in the same time much more arithmetic operations on 16 bit floating-point variables (FP16) than on 32 bit variables. Specifically, performance can be boosted by using Tensor Cores: hardware units prepared to accelerate matrix multiplications, introduced by Nvidia in Volta GPU architecture [7]. So reducing precision could not only reduce memory consumption but also measurably improve time needed for model convergence.

Common technique, used to improve speed, but not to break performance, is to keep model weights in two copies – one in FP32 and one in FP16. In each iteration we would use FP16 one to perform forward, backward propagation, and then to calculate gradients of every weight. Then we would use these gradients to update weights of original, FP32 model weights. Finally, we would make a new instance of FP16 model weight by cutting off precision of updated FP32 model [8].

Another heuristic is to split operations on tensors into two groups. In the first groups we would place for instance transitions through fully connected and convolutional layers. In general computational heavy and precision safe operations. In the second group we would place all the other operations: for instance softmax and batchnorm. Now we would port to FP16 operations from the first group and keep in FP32 operations from the second group. This way we would be safe from precision issues and still achieve pretty good speedup (in comparison to the model, that performs all computations in FP16) [8].

There is one more thing, we need to care about, while reducing precision. FP16 uses only 5 bits for exponent, and its representable range is quite narrow, so that in some cases gradients are small enough to be eventually rounded to 0. To prevent that, one may multiply loss by a power of two, then perform backward propagation algorithm, and then scale back obtained gradients. This solution works pretty well in practice. However now we have to deal with another problem. Gradients calculated in the process may overflow. And even one overflow unables us to calculate gradients for parameters depended on that particular weight. So the scaling factor, rather than being one fixed power of two, need to be adjusted during training: decreased after detection of overflow and increased if too many gradients are zeroed

as a result of rounding [11].

Speaking from more practical side, we used Apex AMP, which is a PyTorch extension developed to ease mixed precision training. It supports automatic dynamic loss scaling, and both described above ways of performing mixed precision training.

Further description of this tool can be found in [9].

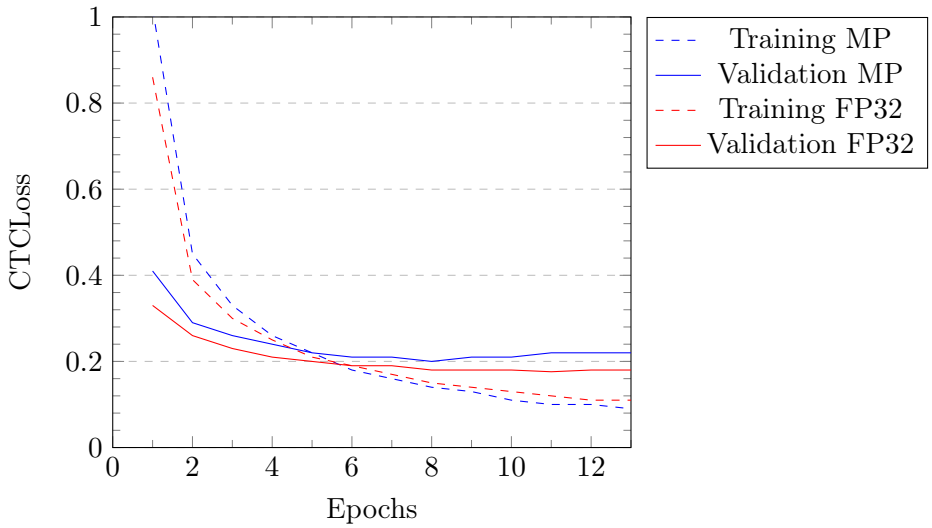

Figure 2.1: Comparison of training with and without AMP.

Figure 2.1 presents comparison of training process with and without AMP, denoted as MP (standing for Mixed Precision) and FP32 respectively. We used the same architecture and the same hyperparameters in both cases. One epoch in FP32 lasted for around 135 minutes, while in mixed precision only 75 minutes. Therefore, we got 1.8 speedup. When it comes to the performance, one can see from the figure 2.1 that the training loss was falling more or less in the same way. To measure models performance we calculated the validation loss - loss function on all examples from the validation dataset. The reason for this is that computing the validation loss is less computationally expensive then WER. We found that validation loss was always a bit lower for FP32 training. Figure 2.2 shows that FP32 model was a bit better also in WER as final WER for FP32 was 13.1% and for MP it was 14.4%. Looking at validation loss, we can conclude, that we can use it as a convenient metric to know when we should stop trainig and we can calculate loss function only after training is finished. What's more, we decided to run rest of experiments only with AMP as it gives us comparable results as usual FP32 training and is much faster. **Z uwagi na nasz błąd te liczby na wykresie będą jeszcze raz policzone i nieznacznie się zmienią. Co więcej, w kolejnych podrodziałach liczba 12.77% również ulegnie zmianie – pozostałe będą takie same.**

2.2. Language model and decoding predictions

***N*-gram language model** is a model of a language in which we assign probabilities of occurrence to sequences of N words based on some large unlabelled text data. Incorporating model to prediction decoding helps avoid typos which are usually phonetically plausible and often occur on words that rarely or never occur in training set. We use Librispeech 4-gram language model available in [10].

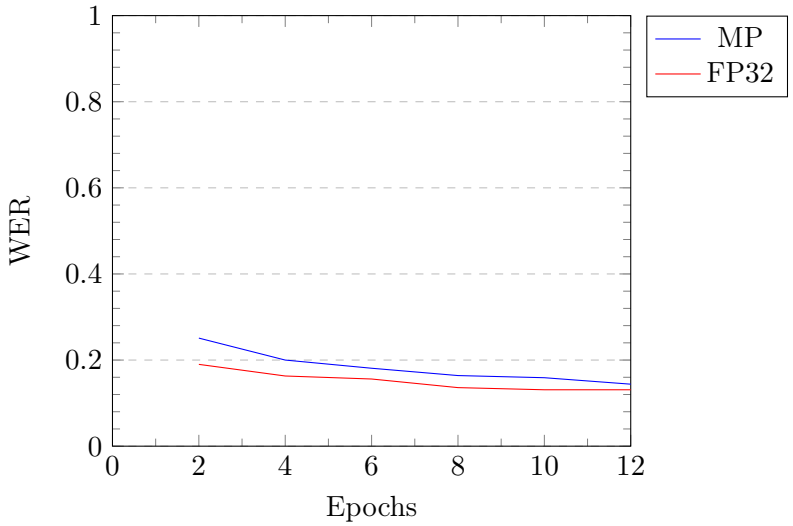

Figure 2.2: Comparison of WER with and without AMP.

Performances with the language model in comparison to no language model are 12.77% WER and 13.35% WER.

At the beginning we also experimented with an easier language model that assigned probabilities to sequences of letters as it was much smaller and simpler. Unfortunately WER was increasing after applying such LM so we abandoned this idea for the bigger and more reliable N-gram LM.

2.3. Regularization

2.3.1. Batch normalization

Batch normalization is a technique for improving the speed, performance, and stability of artificial neural networks, primarily introduced to reduce internal covariate shift [16]. Method based on the idea of normalizing data within each mini-batch using stochastic optimizations for mean and variance.

Denote by B mini-batch of a training set, by m size of B. If input is d–dimensional: $x = (x^{(1)}, x^{(2)}, \ldots, x^{(d)})$, we calculate mean and variance as follows:

$$\mu^{(k)} = \frac{1}{m} \sum_{i=1}^{m} x_i^{(k)}$$

$$\sigma^{(k)^2} = \frac{1}{m-1} \sum_{i=1}^{m} \left(x_i^{(k)} - \mu^{(k)} \right)^2 .$$

Next step is normalization:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu^{(k)}}{\sqrt{\sigma^{(k)^2} + \epsilon}}$$

where ϵ is small constant added for numerical stability. Finally we scale and shift normalized data:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

where $\gamma^{(k)}$ and $\beta^{(k)}$ are learnable parameters.

In our model we use batch normalization after each convolutional layer. When we tried training our model without batch normalization layers, we found out that model didn't learn. Training loss was very high, over 2.5, and validation loss was NaN. That experiment clearly shows that batch normalization is crucial for proper model training and without it we can't expect model to work correctly.

2.3.2. Dropout

Dropout is regularization strategy patented by Google [18], it is used to remove neural network overfitting. The basic idea of this technique is multiplying input and hidden units, during training process, by randomly generated bit mask (or in other words with a given probability we decide whether or not given neuron will be used in the next iteration) [17]. It follows that some nodes will be inactive, set to 0, and model is prevented from complex co-adaptations.

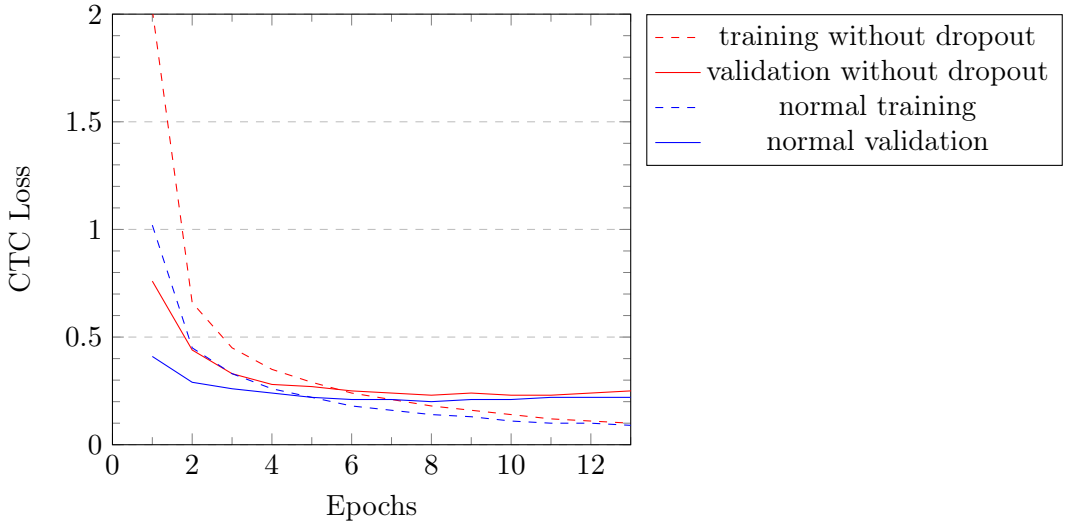

Figure 2.3: Dropout plot

Figure 2.3 presents results of our experiments with and without dropout. One can see that dropout has the biggest impact in the beginning of the training but as the time goes by, results of both versions are getting similar. However, if one don't have enough time to train model for multiple epochs, dropout is necessary to achieve good results. Our final WER results are 12.77% WER for net with dropout and 17% WER for net without dropout.

2.3.3. L2 regularization

$L2$ regularization is a technique designed to prevent neural network from overfitting training dataset. The key idea behind this method is keeping network weights from growing too large unless it is really necessary. It can be realized by adding a term $\lambda||W||_F^2$ to the cost function that penalizes large weights, where $||W||_F$ is a Frobenius norm, and λ is parameter governing how strongly large weights are penalized [15].

Performed experiments showed that applying L2 regularization does not have positive effect on loss function convergence. With weight decay scale equal to 0.0005 CTC loss oscillated around 1.

2.4. Recurrent unit type

In section 1.2 we mentioned bidirectional recurrent layers. Three most commonly used ones are vanilla **Recurrent Neural Networks** (RNN) [12], **Long Short Term Memory Unit** (LSTM) [13] and **Gated Recurrent Unit** (GRU) [14]. We will deliver a brief description of these units based on [2].

Let's denote by l the layer number. A bidirectional recurrent layer h^l consist of a forward recurrent layer $\overrightarrow{h^l}$ and a backward recurrent layer $\overleftarrow{h^l}$. The forward and backward recurrent layer activations for time t are computed as $\overrightarrow{h_t^l} = g(\overrightarrow{h_{t-1}^l}, h_t^{l-1})$ and $\overleftarrow{h_t^l} = g(\overleftarrow{h_{t+1}^l}, h_t^{l-1})$. In the end, we add both partial layers to get $h^l = \overrightarrow{h^l} + \overleftarrow{h^l}$.

Function g mentioned before depends on specific recurrent unit type. For vanilla RNN it is just

$$\overrightarrow{h_t^l} = f\left(W^l h_t^{l-1} + \overrightarrow{U^l}\overrightarrow{h_{t-1}^l} + b^l\right)$$

and analogically for the backward recurrent layer. Here W^l and $\overrightarrow{U^l}$ are just weight matrices, b^l is a bias vector and f is an activation function (e.g. ReLU or tanh). For LSTM and GRU function g is much more complex to simulate some kind of *memory*.

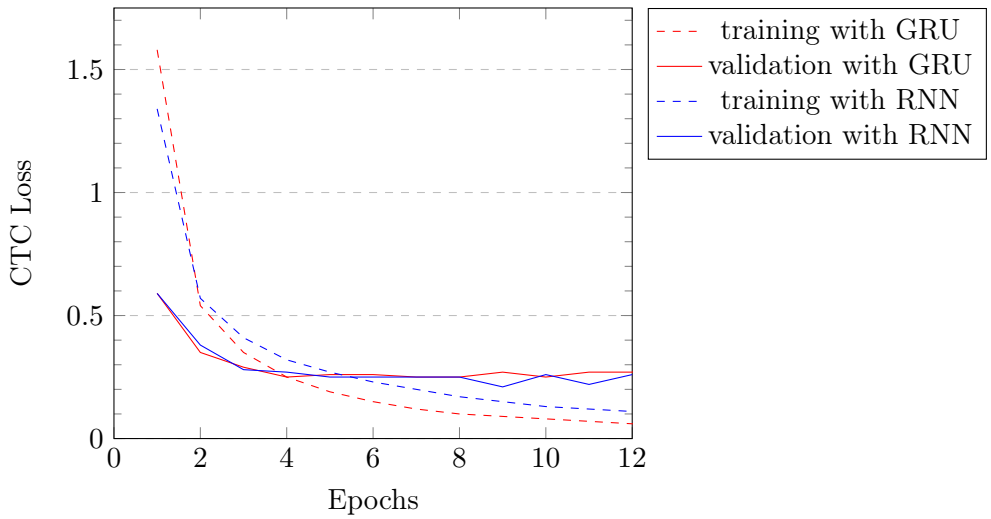

Figure 2.4: RNN vs GRU plot

As we can see in the figure 2.4 there is a noticeable difference in training loss of GRU and RNN layers. In fact, GRU layer achieved WER of 17.45% while RNN layer achieved 12.77%.

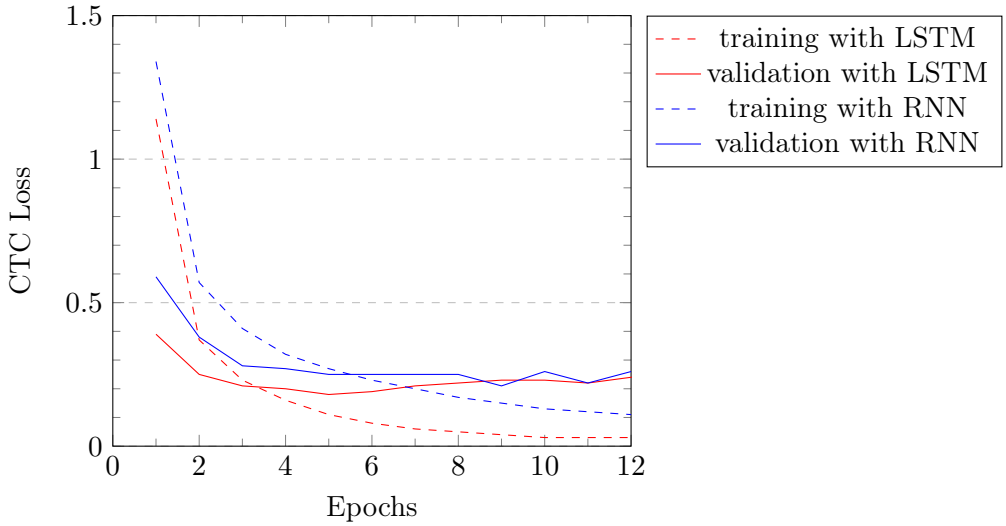

Figure 2.5: RNN vs LSTM plot

Validation loss with LSTM layer appeared to convergence faster than validation loss with RNN layer. In fact, it achieved better WER. Nevertheless we decided to use vanilla RNN as it's faster – it is shown in table 2.1.

RNN	GRU	LSTM
35	53	43

Table 2.1: Epoch training time in minutes

WER results with GRU and LSTM are 17.45% and 13.46% respectively compared to 12.77% of vanilla RNN.

2.5. Performance and multi GPU scaling

For training, we used machine with Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, (which has 20 physical cores), and 256 GB of RAM. It was equipped with four Nvidia Tesla V100 GPGPUs (32 GB of global memory each), connected by NVlink.

Table 2.5 describes performance and scaling we achieved by utilizing more than one GPUs. Performance is measured by time in seconds it took for train for one epoch on full dataset of 960 hours.

number of GPUs	time in minutes	speedup
1 GPU	119	1
2 GPUs	75	1.57
4 GPUs	35	3.37

Table 2.2: Epoch training time comparison

For 11 epochs long mixed precision training on 4 GPUs standard deviation of an epoch time was approximately 14 seconds. It shows that network is scalable, and can be extended to more GPUs.

2.6. Sortagrad and dataset sorting

Training examples that we use vary in length. Longer examples may be described as more challenging. Additionally, as CTC cost function is a negative logarithm of probability, it

gets bigger for longer utterances. Authors of Deep Speech 2 also suspect that these longer utterances usually have bigger gradients. Their proposition for challenging varying lenght of examples is **SortaGrad**. Instead of training on minibatches in random order, in the first epoch we sort them in the increasing order of longest utterance of minibatch. From the second epoch they are shuffled again.

Unfortunately, Apex AMP only supports shuffling dataset for distributed training. Therefore we were not able to test, how sortagrad would affect our results.

2.7. Xavier initialization

We use Xavier initialization [19] for weights in our model. It is a heuristic random initialization of trainable weights in a neural network that assigns values sampled from normal distribution

$$\mathcal{N}\left(0, \sqrt{\frac{2}{N_{in} + N_{out}}}\right)$$

where N_{in} is number of inputs and N_{out} is number of outputs of a certain layer. Parameters of the distribution are chosen in such a way that variances of input to Xavier - intialized layer and output of that layer differ slightly. This prevents exploding and diminishing values during both forward and backward pass through the network and thus speeds up learning.

This is a comparison of training loss with Xavier initialization and with default PyTorch initialization:

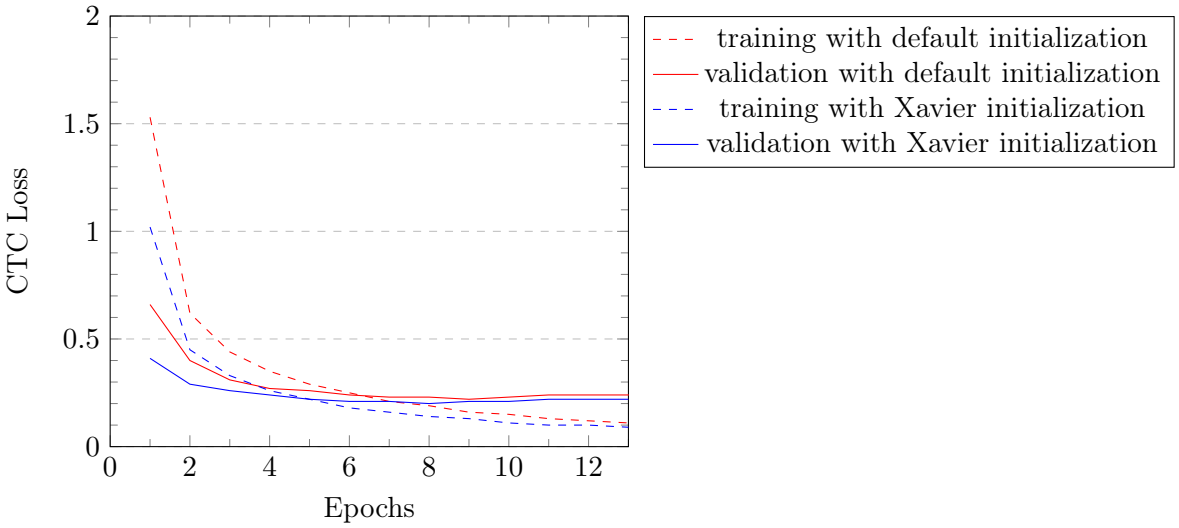

Figure 2.6: Xavier initialization plot

One can easily see in the figure 2.6 that Xavier initialization outperforms default PyTorch initialization in the early stage of training. After around 10 epochs both models have more or less the same training and validation loss – it means that when we train model for a longer time benefits of a proper initialization are not so important. Final WER in both cases is 12.77% and 16.23% for model with Xavier initialization and no initialization respectively.

2.8. Training dataset

As our dataset we use "LibriSpeech ASR corpus" [20] - public ASR dataset containing 960 hours of audiobook English speech for training together with test datasets. This dataset is

much smaller than the dataset used in original Deep Speech 2 paper. Unfortunately original Deep Speech 2 dataset is not publicly available and almost impossible to reproduce with limited resources. However, many other works rely on Librispeech as their dataset [21].

We report our results on model trained on 460 hours of "clean" speech and 960 hours of combined 460 hours of "clean" speech and 500 hours of "other" speech:

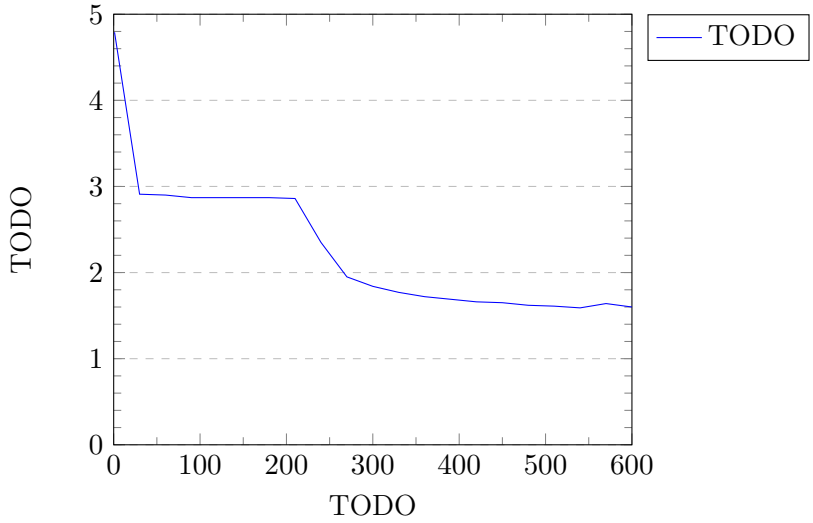

Figure 2.7: TODO

Final WER is **TODO** for model trained on 460 hours dataset and 12.77% for model trained on 960 hours dataset.

2.9. Hyperparameters

To achieve the best accuracy and performance we had to determine what should be values of hyperparameters (e.g. learning rate, batch size, convolutional kernel size, number of features in fully connected layer etc.). We did it by training our network many times with various combinations of mentioned hyperparameters. We looked at results and, according to our intuitions, tried to modify them to achieve better accuracy. What we present in final training scripts is the best combination we found. Probably, there are better ones, but what we have allows us to train model and achieve state of the art accuracy.

Chapter 3

Conclusions

To sum up, we present `PyTorch` scripts for training DeepSpeech2 model for ASR. We also provide already trained models as well as the results of our experiments justifying using specific architecture solutions.

The best model we managed to train achieved 13.1% WER after training for **??** hours. One should notice that original Deep Speech 2 model trained on 1200h dataset achieved 13.8% WER [2], but it used dataset with conversations as well as monologues (and we used monologues only) – that probably explains our better performance. Nevertheless, we can state that we managed to implement and train state of the art model for ASR.

We also managed to speed-up training using potential of GPUs – we trained our model in mixed precision and used multiple GPUs for it. We also showed how exactly that optimizations speed-up training process.

In the future we plan to use prepared model for recognizing other languages. The main problem is finding appropriate datasets – we want ones that are free and have sufficiently short utterances. If we don't find such dataset we can try generating it on our own (e.g. splitting audio-books into shorter parts). We assume that probably we will have to change some hyperparameters but the core architecture should stay the same.

Bibliography

[1] Hannun et al. *Deep Speech: Scaling up end-to-end speech recognition*, Silicon Valley AI Lab 2014, https://arxiv.org/abs/1412.5567

[2] Baidu Research *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*, Silicon Valley AI Lab 2015, https://arxiv.org/abs/1512.02595

[3] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber. *Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In ICML, pages 369-376. ACM, 2006.*

[4] A. Y. Hannun, A. L. Maas, D. Jurafsky, and A. Y. Ng. *First-pass large vocabulary continuous speech recognition using bi-directional recurrent DNNs.* abs/1408.2873, 2014. http://arxiv.org/abs/1408.2873

[5] Bohouta, Gamal & Këpuska, Veton. (2017). *Comparing Speech Recognition Systems (Microsoft API, Google API And CMU Sphinx).* Int. Journal of Engineering Research and Application. 2248-9622. 20-24. 10.9790/9622-0703022024.

[6] https://web.stanford.edu/ jurafsky/slp3/3.pdf

[7] https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/

[8] https://devblogs.nvidia.com/apex-pytorch-easy-mixed-precision-training/

[9] https://nvidia.github.io/apex/amp.html

[10] Librispeech language model http://www.openslr.org/resources/11/

[11] https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html

[12] M. Schuster & K. K. Paliwal. *Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11):2673–2681, 1997* https://pdfs.semanticscholar.org/4b80/89bc9b49f84de43acc2eb8900035f7d492b2.pdf

[13] S. Hochreiter & J. Schmidhuber. *Long short-term memory. Neural Computation, 9(8):1735—1780, 1997* https://www.bioinf.jku.at/publications/older/2604.pdf

[14] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, Yoshua Bengio *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches* https://arxiv.org/pdf/1409.1259.pdf

[15] Anders Krogh, John A. Hertz *A Simple Weight Decay Can Improve Generalization* https://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf

[16] Sergey Ioffe, Christian Szegedy *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* https://arxiv.org/pdf/1502.03167.pdf

[17] David Warde-Farley, Ian J. Goodfellow, Aaron Courville, Yoshua Bengio *An empirical analysis of dropout in piecewise linear networks* https://arxiv.org/abs/1312.6197

[18] System and method for addressing overfitting in a neural network https://patents.google.com/patent/US9406017B2/en

[19] Xavier Glorot, Yoshua Bengio *Understanding the difficulty of training deep feedforward neural networks* http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[20] LibriSpeech ASR corpus http://www.openslr.org/12/

[21] Jayadev Billa, *Improving LSTM-CTC based ASR performance in domains with limited training data* https://arxiv.org/pdf/1707.00722.pdf

All the files were downloaded on June 23, 2019