ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Explicit Word Density Estimation for Language Modelling

Master Thesis

Jovan Andonov

September 15, 2019

Advisors: Prof. Dr. T. Hofmann, PhD. O. Ganea, PhD. G. Bécigneul, PhD. P. Grnarova
Department of Computer Science, ETH Zürich

Abstract

Language Modelling has been a central part of Natural Language Processing for a very long time and in the past few years LSTM-based language models have been the go-to method for commercial language modeling. Recently, it has been shown that when looking at language modelling from a matrix factorization point of view, the final Softmax layer limits the expressiveness of the model, by putting an upper bound on the rank of the resulting matrix. Additionally, a new family of neural networks based called NeuralODEs, has been introduced as a continuous alternative to Residual Networks. Moreover, it has been shown that there is a connection between these models and Normalizing Flows. In this work we propose a new family of language models based on NeuralODEs and the continuous analogue of Normalizing Flows and manage to improve on some of the baselines.

Acknowledgements

First and foremost, I would like to thank my family for always believing in me.

Next, I would like to thank my supervisors Octavian, Gary and Paulina for all the useful discussions. I would also like to thank the Data Analytics Lab and the Leonhard cluster for GPU access and free coffee.

A big thank you goes to Gorjan and Ondrej for all their advice, useful discussions and comments on the thesis text. I would also like to thank Igor for his constant support.

Last, but not least, I would like to thank Tina for the never-ending support and for always being there for me.

Contents

Chapter 1

Introduction

Anything we see or do, can be described and contained within a sequence of words, meaning that the entire complexity of the world can be embedded in a piece of text. This is exactly what makes text and textual communication so important in our daily lives. Additionally, this is also what makes text processing and textual communication so complex for machines. Namely, the area of Computer Science that deals with these problems is called Natural Language Processing (NLP). NLP is a vast area with many subcategories, but without doubt, one of its core and most vital subcategories is text understanding and text generation.

In NLP, the tools that are used for text generation are called Language Models (LM). Let's consider the following sentence:

Look at all those clouds, it is going to ...

Given the sentence above as a context, a Language Model will then try to estimate what is the most likely word to end the sentence. From a mathematical point of view, LMs try to learn a context conditioned probability distribution over a vocabulary. This means that given a vocabulary of available words and a sequence of words that represents the history or the context, a Language Model processes the context and returns a discrete probability distribution over the vocabulary

$$P(w|w_{1..i-1}).$$

Here $w_{1..i-1}$ is the context, often denoted as h, and w is a discrete random variable that represents the vocabulary. We can then proceed with generating the next word by simply selecting the word with highest probability as

$$\hat{w} = argmax_w P(w|w_{1..i-1})$$

However, very often we do not want to only generate a single word given a context, but instead we want to generate whole sequences. Therefore, the LMs can also be seen as tools that model the joint probability distribution over a textual sequence. Or mathematically speaking

$$P(w_1, ..., w_n) = \prod_i^n P(w_i|w_{1..i-1}).$$

First LMs were count based and called N-grams [20]. However, with the recent advances in Deep Learning, LMs based on Neural Networks are currently dominating the field. Since the first Neural Language Model [1] which was based on Feedforward Neural Networks, things have evolved and now Recurrent Neural Networks (RNN) [22] are the standard. Additionally, as neural networks are trained with gradient based methods and back-propagation [30], people have figured out that RNNs, when processing long contexts can suffer from the vanishing or exploding gradients problem [13, 25, 26]. Therefore, Vanilla RNNs were substituted with Long short-term Memory (LSTM) [14] based RNNs. LSTMs alleviate the vanishing gradient problem and additionally gradient clipping [26] takes care of the exploding gradients problem. Recently, Transformer [34] based models like BERT [4] and GPT-2 [28] have enjoyed quite the success in language modelling and language understanding tasks. However, these models have an enormous number of parameters and need an enormous amount of resources to be trained. Therefore, in the past few years, LSTM-based RNNs with a Softmax layer on top have been the go-to method for commercial language modelling.

This thesis first gives an overview of the current limitations of Language Modelling. Then it describes how previous work has tried to break these limitations. Then, introduces a novel idea for overcoming the previously mentioned limitations of Language Modelling. Towards the end, it describes the architecture of the models created in the scope of this thesis, as well as present the results from the experiments. Finally, it concludes the findings and suggests possible future work.

Chapter 2

Current Limitations of Language Models

2.1 The Softmax Bottleneck

The majority of parametric LMs use a Softmax function operating on the context c, and a word embedding e_w to define the conditional distribution $P_\theta(w|c)$, where w stands for a word, and θ are the parameters of the model. More specifically, the model distribution is usually written as

$$P_\theta(w|c) = \frac{h^T e_w}{\sum_{e_{w'}} h^T e_{w'}},$$

where h is a function of c and it is commonly obtained using a Recurrent Neural Network, and e_w is the embedding for word w. Additionally, $h \in R^D$ and $e_w \in R^D$ and both of them depend on θ. Finally, we refer to the dot product $h^T e_w$ as a *logit*.

Even though, one might argue that natural languages contain an infinite amount of contexts, let's take a look at the finite case first. We can assume that a natural language consists of N contexts and M words. Consequently, we can describe Language Modelling as a matrix factorization problem. Consider the following matrices:

$$H_\theta = \begin{bmatrix} h_{c_1}^T \\ h_{c_2}^T \\ \vdots \\ h_{c_N}^T \end{bmatrix} \quad E_\theta = \begin{bmatrix} e_{w_1}^T \\ e_{w_2}^T \\ \vdots \\ e_{w_M}^T \end{bmatrix}$$

$$A = \begin{bmatrix} \log P(w_1|c_1) & \log P(w_2|c_1) & \dots & \log P(w_M|c_1) \\ \log P(w_1|c_2) & \log P(w_2|c_2) & \dots & \log P(w_M|c_2) \\ \vdots & \vdots & \ddots & \vdots \\ \log P(w_1|c_N) & \log P(w_2|c_N) & \dots & \log P(w_M|c_N) \end{bmatrix}$$

Where $H_\theta \in R^{N \times D}$ is a matrix containing the hidden states for every context as row vectors. $E_\theta \in R^{M \times D}$ is a matrix containing word embeddings for every word as row vectors. A is a matrix containing the true log probabilities of every word, given every context. Then, language modelling can be described as:

$$H_\theta E_\theta^T = \hat{A}$$

Where, $\hat{A}$ is:

$$\hat{A} = \begin{bmatrix} \log P_\theta(w_1|c_1) & \log P_\theta(w_2|c_1) & \dots & \log P_\theta(w_M|c_1) \\ \log P_\theta(w_1|c_2) & \log P_\theta(w_2|c_2) & \dots & \log P_\theta(w_M|c_2) \\ \vdots & \vdots & \ddots & \vdots \\ \log P_\theta(w_1|c_N) & \log P_\theta(w_2|c_N) & \dots & \log P_\theta(w_M|c_N) \end{bmatrix}$$

and we want it to be as close as possible to the true A. Now we can ask the following question:

"What is the expressiveness of this language model?"

We can then proceed to answer this question from a matrix factorization point of view. Essentially, we want to learn matrices H_θ and E_θ such that we will be able to factorize the true distribution A. However, in order for a valid factorization to exist the rank of $H_\theta E_\theta^T$ has to be at least as large as the rank of A, i.e. $\text{rank}(H_\theta E_\theta^T) \geq \text{rank}(A)$. As $H_\theta \in R^{N \times D}$ and $E_\theta \in R^{M \times D}$, $\text{rank}(H_\theta E_\theta^T)$ is bounded by D. Therefore, this is a limitation that comes from the final *Softmax layer*. It simply means, that no matter how efficient we are in embedding all contexts into a matrix H_θ, we will not be able to retrieve the true language distribution A, unless $D \geq \text{rank}(A)$.

To realize why this is indeed a bottleneck, and a problem in language modelling, we should first consider the typical dimensionalities that are used for the hidden state and the word embeddings. Usually, D is in the low hundreds, while the rank of the true distribution A can theoretically be up to M which is usually at least as large as 10^4. Right off the bat, we have a mismatch of several orders of magnitudes. One might say that an easy fix is to simply increase D and have a $M \times M$ *Softmax* in the final layer. However,

this will drastically increase the amount of trainable parameters, resulting in slower training and harder optimization. Even though, wider and larger neural networks are theoretically more expressive, in practice they are a lot more difficult to train.

As using a larger D is not a straightforward solution to the problem it means that typical language models are *Low Rank Language Models*. This would only cause problems if the true distribution A is indeed of a high rank. It is very hard to prove that natural languages are of high rank. However, intuitively speaking, if the true distribution of a natural language was indeed to be of a low rank, it would mean that all semantic meanings can be created by combining a small number of meanings. Which seems very odd and no linguist has ever managed to find such a small subset of bases, which can fully describe a language. Therefore, Yang et al. [36] speculate that a high rank language model is needed to capture the true distribution.

2.2 Single Transformation

Another limitation of classical language models is the output projection layer. Taking the matrix-vector product between the embedding matrix (nowadays weights between the output projection and the embedding matrix are usually tied) and the hidden state, further exponentiating and normalizing the result to get a probability distribution, essentially results in a transformation that behaves as a single mode Gaussian around the hidden state. This means that the majority of the probability mass is concentrated on one very small continuous subspace of the embedding space. This subspace usually corresponds to the surroundings of the word embedding that is mostly associated with the given context throughout training. Take the following sentence as an example:

"I want to buy..."

It is easy to see that multiple words are likely to come after this particular context even though they might be associated with different contexts on average. This is a very common scenario in natural languages where the distribution over the vocabulary is considered to be a *fat tail* distribution. For example take the words *car* and *computer*. By training a Language Model and plotting the embeddings, it is possible to visualize that the neighbors of *car* in the embedding space are *truck, auto, bus, vehicle* etc. and the neighbors of *computer* are *desktop, portable, electronic, laptop* etc. Refer to figures 2.1 and 2.2. This indicates that *car* and *computer* are in different neighborhoods of the embedding space.

Now imagine every single word that can finish the previous sentence. Every one of those words is embedded into its own neighborhood where it is close

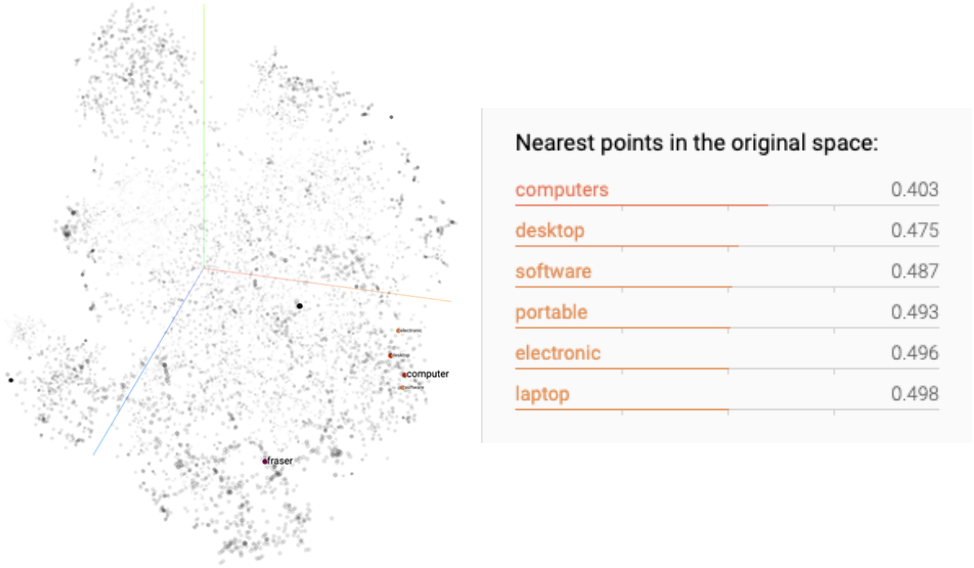

Figure 2.1: *Neighborhood of* car *in the embedding space of AWD-LSTM.*

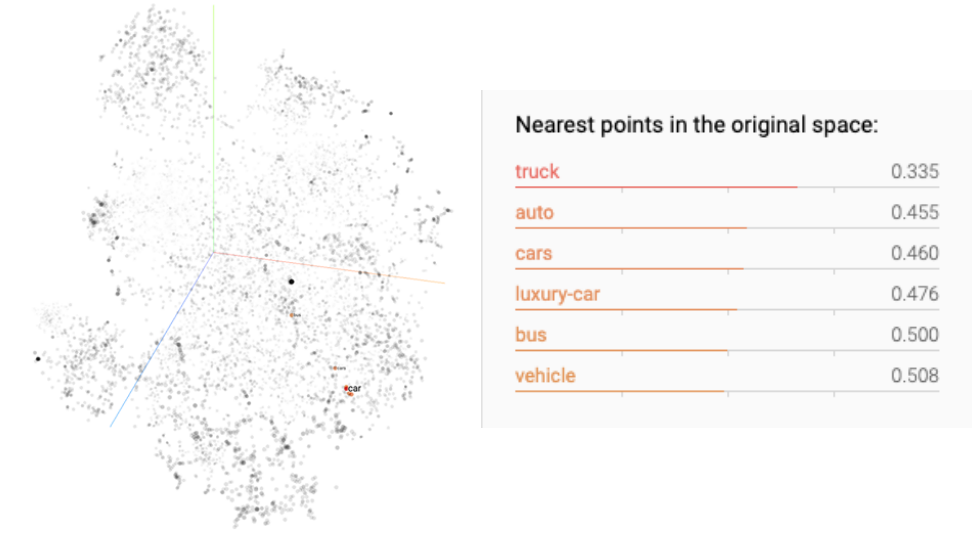

Figure 2.2: *Neighborhood of* computer *in the embedding space of AWD-LSTM.*

to other words that share the same context on average. Therefore, a simple single mode Gaussian seems very limited for all those *"fat tail"* situations in natural languages. Ideally, we would like something that can adapt based on the context, i.e. given a hidden state we can obtain different type of distributions. It is also possible that the success of MoS [36] and DOC [33] papers, in addition to breaking the *softmax bottleneck*, is due to the fact that they learn a dynamic mixture of Gaussians, which by definition is more expressive. Additionally, based on the amount of Gaussians in the mixture, those approaches are more or less capable of solving the previously mentioned issues.

To generalize, it seems unlikely that a single general distribution, conditioned on the hidden state, can capture all possible situations in Language Modelling. Therefore, it seems reasonable that based on the context, or the hidden state if you want, we would like to obtain custom distributions that specifically fit the need of the current context. But what if we can start with a "simple", single, general distribution and then based on the context, distort it into a more complex distribution if needed? That would alleviate the need to manually tune the amount of components in the mixture and can be seen as a generalization of MoS [36] and DOC [33].

Chapter 3

Related Work

3.1 AWD-LSTM

Merity et al. [21] had several crucial contributions to RNN-based language modelling with their AWD-LSTM model.

First, regularizing RNNs is a complicated matter. A naïve application of dropout [32], where we randomly dropout units in every time step, results in unit starvation and disrupts the RNN's ability to retain long term dependencies. Gal and Ghahramani [6] suggest using the same dropout mask in every time step, to prevent this from happening. On the other hand, Merity et al. [21] propose the weight-dropped LSTM, which uses Dropconnect [35] on the hidden-to-hidden weights as a recurrent regularization.

Secondly, they propose the use of Non-monotonically Triggered Averaged Stochastic Gradient Descent (NT-ASGD) as an optimization algorithm, instead of the regular Stochastic Gradient Descent (SGD).

The training of neural networks can be defined as

$$\hat{\theta} = \underset{\theta}{argmin} \frac{1}{N} \sum_{i=1}^{N} f_i(\theta),$$

where f_i is the loss function for the i-th data point and θ are the parameters to be learned. SGD takes the form of

$$\theta_{k+1} = \theta_k + \gamma_k \hat{\nabla} f(\theta_k),$$

where k stands for the iteration number, γ_k is the learning rate and $\hat{\nabla}$ denotes a stochastic gradient on a minibatch of samples. After convergence, SGD returns the final iteration as the solution. Contrary to this, Averaged SGD (ASGD) returns the average

$$\frac{1}{K-T+1}\sum_{i=T}^{K}\theta_i$$

as a solution. Here K is the total number of iterations and $T < K$ is a user-specified averaging trigger. Merity et al. [21] propose to use an automatic trigger mechanism for the averaging. Instead of manually specifying a value for T, they propose a non-monotonic criterion that triggers the averaging when the validation metric does not improve for several epochs. The full method is shown in algorithm 1.

Algorithm 1: Non-monotonically Triggered ASGD (NT-ASGD) [21]

Inputs: Initial parameters θ_0, learning rate γ, logging interval L, non-monotone interval n

Initialize $k \leftarrow 0,\ t \leftarrow 0,\ T \leftarrow 0,\ logs \leftarrow []$

while *stopping criterion not met* **do**
  Compute stochastic gradient $\hat{\nabla} f(\theta_k)$ and take the SGD step
  if *mod(k, L) = 0 and T = 0* **then**
    Compute validation perplexity v
    if $t > n$ *and* $v > \min_{l \in t-n, \ldots, t} logs[l]$ **then**
      Set $T \leftarrow K$
    end
    Append v to $logs$
    $t \leftarrow t + 1$
  end
end
return $\frac{1}{k-T+1}\sum_{i=T}^{k}\theta_i$

Finally, their codebase [1] set a new foundation for RNN-based language modelling. Every other notable model that came out in the next few years used their codebase as a basis to build upon.

3.2 Mixture of Softmaxes

Yang et al. [36] contributions in their paper *Breaking the Softmax Bottleneck* are two-fold. First, they identify the Softmax Bottleneck problem described

[1] https://github.com/salesforce/awd-lstm-lm

in section 2.1 and secondly, they propose a simple technique that allows to bypass this limitation.

They use the AWD-LSTM model to encode the context in a vector g_c. Then, they project g_c to obtain multiple hidden states

$$h_{c,k} = tanh(W_{h,k} g_c)$$

where $h_{c,k}$ stands for the k-th component for context c and W_k are trainable parameters. Then, the conditional probability of a word given a context is modelled as

$$P_\theta(w|c) = \sum_{k=1}^{K} \pi_{c,k} \frac{\exp h_{c,k}^T e_w}{\sum_{w'} \exp h_{c,k}^T e_{w'}}$$
$$s.t. \sum_{k=1}^{K} \pi_{c,k} = 1,$$

where e_w is the (output) word embedding for word w. The prior weights $\pi_{c,k}$ are obtained by

$$\pi_{c,k} = \frac{\exp w_{\pi,k}^T g_c}{\sum_{k'=1}^{K} \exp w_{\pi,k'}^T g_c}.$$

According the previous equations, we can deduce that Yang et al. [36] propose to have a weighted average between several Softmax functions. Therefore, they call this model *Mixture of Softmaxes (MoS)*. Furthermore, if we create a matrix $\hat{A}_{MoS}$ similar to the one in section 2.1, we get

$$\hat{A}_{MoS} = \log \sum_{k=1}^{K} \Pi_k \exp(H_k E).$$

As $\hat{A}_{MoS}$ is now obtained via a non-linear transformation, we can deduce that its rank is not bounded and $\hat{A}_{MoS}$ can potentially be a full-rank matrix.

3.3 Direct Output Connections

Takase et al. [33] propose their *Direct Output Connections (DOC)* model as a generalization of MoS. DOC computes J probability distributions from all layers and performs a weighted average between them. The output probabilities in DOC, are computed as

$$P_\theta(w|c) = \sum_{j=1}^{J} \pi_{j,c}\ Softmax(\tilde{W}k_{j,c})$$
$$s.t. \sum_{j=1}^{J} \pi_{j,c} = 1$$

where $\pi_{j,c}$ is the weight for the j-th component in the mixture given context c and is obtained by

$$\pi_c = Softmax(W_\pi h_c^N),$$

where π_c is a vector with elements $\pi_{j,c}$, W_π is a weight matrix and h^N is the hidden state from the final layer. Furthermore, $\tilde{W} \in R^{|V|\times d}$ is a weight matrix and $k_{j,c} \in R^d$ is a vector computed from the hidden state of some layer n as

$$k_{j,c} = W_j h_c^n.$$

In this equation $W_j \in R^{d\times d_{h^n}}$ is a weight matrix. Additionally, let i_n be the number of k-s computed from the hidden state of the n-th layer s.t. $\sum_{n=0}^{N} i_n = J$. From here we can deduce that for $i_N = J$, i.e. if all distributions are obtained from the final layer, DOC is equivalent to MoS, which is exactly why DOC is considered to be a generalization of MoS.

Finally, if we construct the matrix containing all log-probabilities given all contexts for DOC, we can notice that it takes the form of

$$\hat{A}_{DOC} = \log \sum_{j=1}^{J} \Pi\ Softmax(K_j \tilde{W}^T)$$

where Π is a diagonal matrix whose entries are the weights $\pi_{j,c}$ and K_j is a matrix whose rows are vectors $k_{j,c}$. As $\hat{A}_{DOC}$ is obtained using a non-linear transformation, $\hat{A}_{DOC}$ can be of an arbitrary high rank and is not limited by the Softmax Bottleneck explained in section 2.1.

Chapter 4

Neural Ordinary Differential Equations

4.1 Introduction to Neural ODEs

In recent years Residual Networks (ResNet) [12] have brought a great success in Deep Learning and especially in computer vision. They have proven to be effective against the vanishing gradient and the degradation problems and have drastically eased the optimization of very deep neural networks. If we refer to the output vector of each layer as z_t where t stands for the layer, then Residual Networks can be mathematically described as

$$z_{t+1} = z_t + f(z_t;\ \theta_t), \tag{4.1}$$

where $t \in \{0, ..., T\}$, $z_t \in R^D$ and θ_t are the parameters of the t-th layer. These iterative updates can be interpreted as an Euler discretization of a continuous transformation [19, 11, 31].

Moreover, as we add more layers and take smaller steps, in the limit, we parameterize the continuous dynamics of the hidden state using an ordinary differential equation (ODE) specified by a neural network

$$\frac{dz(t)}{dt} = f(z(t),\ t;\ \theta). \tag{4.2}$$

Chen et al. [3] introduced this concept as a new family of deep neural network models, where the neural network outputs the gradient of the hidden state with respect to the depth. Then, given an initial state and the differential equation parameterized by the neural network, the final state is obtained by solving an ODE. The analogy they make is the one that considers this family of models to be the continuous case of ResNets. Figure 4.1, depicts the similarities and differences between ResNets and neural networks based

on ODEs. They call this family of models *Neural ODEs* or *ODENets*, and provide an open source framework implemented in PyTorch[1].

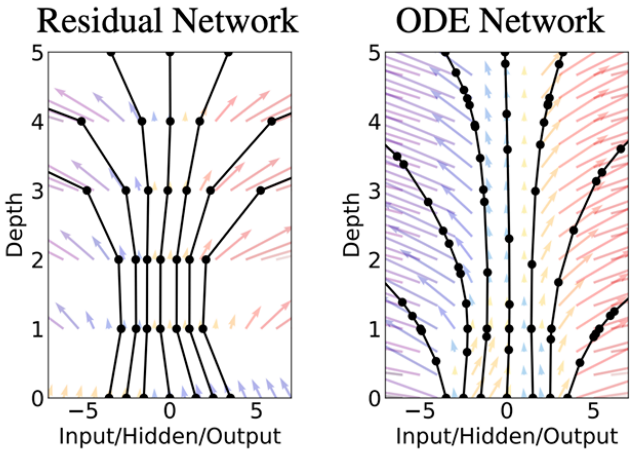

Figure 4.1: Left*: A Residual network defines a discrete sequence of finite transformations.* Right*: An ODE network defines a vector field that continuously transforms the state [3].*

In equation 4.2, $z(t)$ is the hidden state in the t-th layer, and f can be any neural network parameterized by θ, with $z(t)$ and t as inputs and the gradient of $z(t)$ with respect to t as output. Furthermore, the final output of such a model can then be defined as

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} \frac{dz(t)}{dt}\, dt \tag{4.3}$$

$$= z(t_0) + \int_{t_0}^{t_1} f(z(t),\ t;\ \theta)\, dt \tag{4.4}$$

$$= ODESolve(z(t_0), f, t_0, t_1). \tag{4.5}$$

According to the equations above, we can conclude that, f is learning a vector field, which is why Neural ODEs can potentially be seen as models with infinite amount of layers. To be specific, the number of layers is dynamically decided and delegated to the ODE solver. Furthermore, Chen et al. [3] developed their framework in way that any ODE solver can be used as a blackbox. This allows for more flexibility and decouples the framework from the ODE solver. General purpose ODENets are illustrated on Figure 4.2.

Defining neural network models in this fashion has several advantages [3]:

[1] https://github.com/rtqichen/torchdiffeq

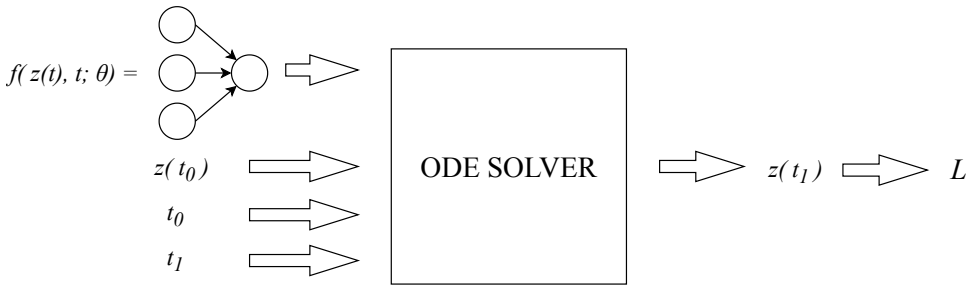

Figure 4.2: *General purpose ODENet. $f(z(t),\ t;\ \theta)$ is neural network specifying the differential equation, $z(t_0)$ is the initial state, t_0 is the initial time, t_1 is the final time, $z(t_1)$ is the final state and L is a scalar valued loss function.*

- **Memory Efficiency.** In section 4.2 it is discussed how circumventing backpropagation through the operations of the ODE solver saves a lot of memory.
- **Adaptive Computation.** Nowadays, ODE solvers provide guarantees about the growth of the approximation error, monitor the error, and adapt their evaluation strategy on the fly to achieve the required level of accuracy. This allows for explicit control over the speed versus precision trade-off.
- **Parameter Efficiency.** In section 3 of their work, Chen et al. [3] demonstrate how this family of models ties the weights of nearby layers, resulting in fewer parameters without the loss of performance.
- **Scalable and invertible normalizing flows.** Chapter 5 sshows how one side effect of going in the continuous domain, allows for an easier and unrestricted use of normalizing flows. As a result, normalizing flows can be used for language modelling, as shown in chapter 6.
- **Continuous time-series models.** RNNs are the de-facto architecture for time-series models. Unfortunately, they require that the observations are discretized and bound to specific emission intervals. On the other hand, continuously defined dynamics can naturally take care of observations that arrive at arbitrary times.

4.2 Backpropagation through the ODE solver

An immediate question that rises, is how does one backpropagate through the ODE solver. In theory one can simply backpropagate through the operations of the solver, however, this has several drawbacks. First, some solvers, require solving a nonlinear optimization problem at every step. This can make direct backpropagation through the integrator difficult. Additionally,

as mentioned in the previous section, ODENets can potentially have a very high number of layers. Backpropagating through such a large number of layers is inefficient from a memory point of view, as it would mean that all intermediate steps should be kept in memory until the backward pass is over. Therefore, what Chen et al. [3] propose, is to compute gradients by solving a second augmented ODE backwards in time. This method is called the *adjoint sensitivity method* [27] and is applicable to all ODE solvers. It scales linearly with problem size, has low memory cost, and allows for explicit control over numerical errors.

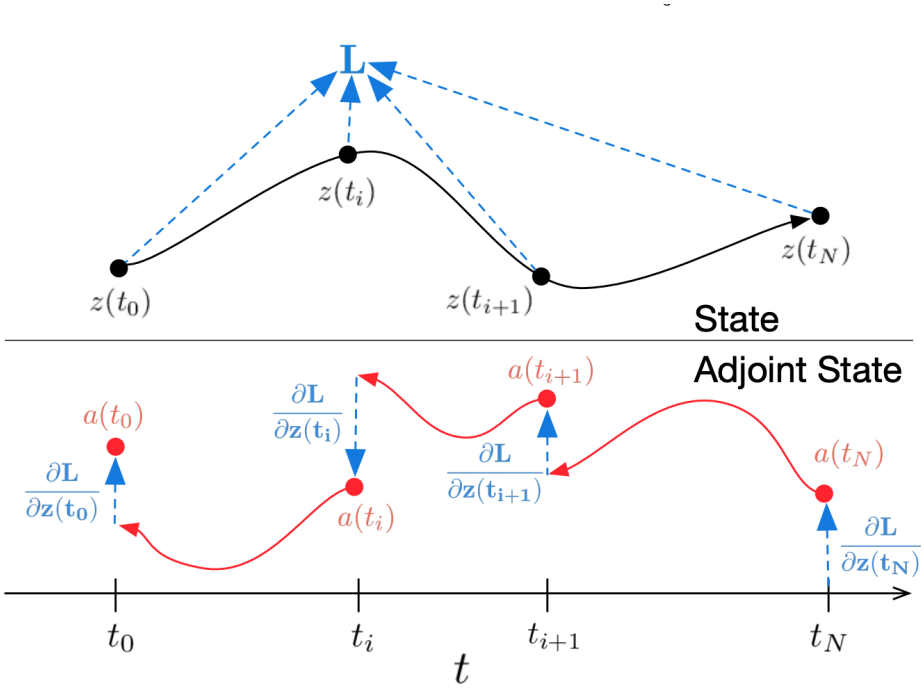

Figure 4.3: *Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The red lines denote the sequence of separate ODE solves [3].*

As shown in figure 4.2, we need to optimize a scalar valued loss function L whose input is the output of the ODE Solver.

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t),\ t;\ \theta)\right) = L(ODESolve(z(t_0), f, t_0, t_1)) \tag{4.6}$$

The parameters of the ODENet are the parameters of the neural network f, which is why we are interested in the derivative of the loss function with

respect to θ. Pontryagin [27] show that the derivative takes the form of another initial value problem

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \left(\frac{\partial L}{\partial z(t)}\right)^T \frac{\partial f(z(t),\ t;\ \theta)}{\partial \theta} dt \tag{4.7}$$

Where $\partial L/\partial z(t)$ is known as the *adjoint state* of the ODE. Chen et al. [3] use one call to an ODE solver to get $z(t_1)$ and then a second one to calculate the equation 4.7. In cases where the loss depends not only on the final state $z(t_1)$, but also on the intermediate states $z(t)$, the reverse-mode derivative must be broken into a sequence of separate solves as shown on figure 4.3.

4.3 Neural ODEs for Language Modelling

We can look at language modelling as a two step process. First, we encode the context into a hidden state vector, and then using the hidden state vector we generate a distribution over the vocabulary. The former is commonly done by an LSTM-based RNN. Then, given the hidden state, a Softmax layer is used to obtain the probability distribution over the vocabulary. In section 2.1, it is discussed how having this Softmax layer introduces a theoretical limitation on LMs, and in sections 3.2 and 3.3 is discussed how models like MoS [36] and DOC [33] have tried to break it. In this section, a simple model based on Neural ODEs is proposed, that is not restricted by the Softmax Bottleneck problem.

Let V be the vocabulary, h be the hidden state, d be the hidden state dimensionality, $L \in R^{d \times |V|}$ be the output projection matrix (if weights are tied [16] this is also the embedding matrix), and $y^* \in R^{|V|}$ be a one-hot encoded ground truth vector. The model can then be represented as

$$\begin{aligned} l(t_0) &= L^T h,\ l \in R^{|V|} \\ l(t_1) &= node(l(t_0),\ f) \\ y &= Softmax(l(t_1)), \end{aligned}$$

where *node* is a NeuralODE block defined as

$$node(l(t_0),\ f) = l(t_0) + \int_{t_0}^{t_1} f(l(t),\ t)dt$$

and f is a neural network parameterizing the gradient of the logits l with respect to time or in this case depth. Moreover, f can be an arbitrary neural network architecture and one possibility is to define it as

$$f(l,t) = H_f ReLU(W_f^T l),\ W_f, H_f \in R^{|V|\times|V|}.$$

An apparent problem with this approach lies in the $W_f, H_f \in R^{|V|\times|V|}$ matrices. As the vocabulary can often be in the tens of thousands, this highly increases both the memory and the time complexity of the overall model. However, this problem can be solved by using a dimensionality bottleneck:

$$f(l,t) = H_f ReLU(W_f^T l),\ W_f, H_f \in R^{|V|\times k},$$

where k is the dimensionality of the bottleneck and is usually in the low hundreds.

In the simplest form of f, t can be ignored. This is the same as stating that the gradient does not depend on it and we have a constant vector field as we move through time. Other possibilities are to concatenate it to the input or use it in a conjunction with Hypernetworks [10] to generate W_f and H_f based on t. Both approaches are suggested in Chen et al. [3] and Grathwohl et al. [8]. Finally, we can use *Cross-Entropy* for training.

The main idea behind this approach is to solve the Softmax Bottleneck by applying non-linear transformations on the logits, similarly to what is done in [7]. The difference between the two approaches lies in the nature of the non-linearities applied. Ganea et al. [7] apply monotonic pointwise non-linearities, and here we use a Neural ODE.

Chapter 5

Continuous Normalizing Flows

5.1 Introduction to Normalizing Flows

Section 2.2 ends with a question: "What if we can start with a simple distribution and distort it into a more complex one?". To answer this question, let us first examine what happens to the densities as we transform some random variable.

Let $x \in R^d$ be a random variable with an underlying probability density function $P_X(x)$ and $f : R^d \mapsto R^d$ be an invertible transformation. Then, if

$$y = f(x),$$

i.e. we obtain the random variable y by transforming x using f, the probability density function $P_Y(y)$ can be obtained using the change of variables formula

$$P_Y(y) = P_X(x) \left| \det \frac{\partial f}{\partial x} \right|^{-1} \tag{5.1}$$

and the change in log density becomes

$$\log P_Y(y) = \log P_X(x) - \log \left| \det \frac{\partial f}{\partial x} \right|. \tag{5.2}$$

Now let us assume that instead of a single transformation, we want to apply a series of transformations. Let $f_i : R^d \mapsto R^d$, $i \in \{1, ..., n\}$ be n different transformations, and let z_0 be an initial random variable with a probability density function P_{Z_0}. Then, we can denote the composition of functions f_i on z_0 as

$$z_n = f_n \circ ... \circ f_1(z_0),$$

with the probability density function of z_n being

$$P_{Z_n}(z_n) = P_{Z_0}(z_0) \prod_{i=1}^{n} \left| \det \frac{\partial f_i}{\partial z_{i-1}} \right|^{-1} \tag{5.3}$$

and the total change in log density being

$$\log P_{z_n}(z_n) = \log P_{z_0}(z_0) - \sum_{i=1}^{n} \log \left| \det \frac{\partial f_i}{\partial z_{i-1}} \right|. \tag{5.4}$$

This technique is called a normalizing flow and was formalized by Rezende and Mohamed [29]. They start with a simple probability density function and transform it into a more complex one, by applying a sequence of invertible transformations until a desired level of complexity is obtained. Some simple normalizing flows introduced in their paper [29] are the *planar* and the *radial* flow. The transformation for the planar flow is

$$f(z) = z + uh(w^T z + b),$$

where $u, w \in R^d, b \in R$ and h is a smooth element-wise non-linearity. On the other hand, the transformation of the radial flow is

$$f(z) = z + \beta h(\alpha, r)(z - z_0),$$

where $z_0 \in R^d$, $\alpha \in R^+$, $\beta \in R$, $r = |z - z_0|$ and $h(\alpha, r) = 1/\alpha + r$. The planar flow introduces hyperplanes into the space, and the radial flow introduces spheres into the space.

However, in practice, normalizing flows are limited due to their high computational complexity. By looking at equations 5.1 up to 5.4, we can deduce that normalizing flows require calculating a determinant, which is generally a $\mathcal{O}(d^3)$ operation. Therefore, their expressiveness is limited by the need to use relatively simple transformations, which Jacobians are easy to compute. For example, both the planar and radial flow allow for linear cost determinant computation and their effect are illustrated on figure 5.1.

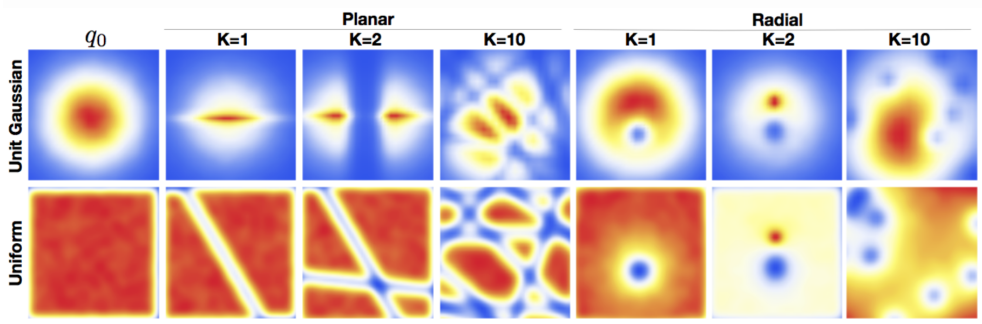

Figure 5.1: *The effect of planar and radial flows on uniform and unit Gaussian distributions [29].*

5.2 Continuous Normalizing Flows

The previous chapter introduces a novel family of neural models under the name Neural ODEs. Chen et al. [3] noticed that the discretized equation 4.1 also appears in Normalizing Flows [29] and the NICE framework [5]. They further realized that performing continuous transformations has an unexpected side effect to the change of variables formula.

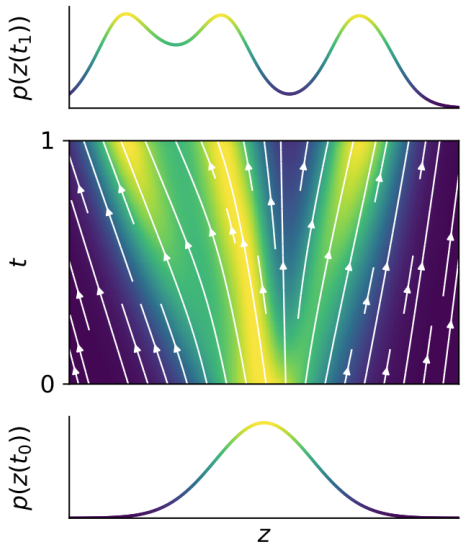

Figure 5.2: *Continuous Normalizing Flows distorting a single mode Gaussian into a more complex distribution [8]*

Recall from equation 5.2, that when applying discrete transformations, the change in log density is given by

$$z_1 = f(z_0), \tag{5.5}$$

$$\log p(z_1) = \log p(z_0) - \log \left| \det \frac{\partial f}{\partial z_0} \right|, \tag{5.6}$$

however, when going into the continuous domain, the change in log density becomes

$$\frac{\partial z(t)}{\partial t} = f(z(t), t), \tag{5.7}$$

$$\frac{\partial \log p(z(t))}{\partial t} = -Tr\left(\frac{\partial f}{\partial z(t)}\right), \tag{5.8}$$

with the total change in log density given by

$$\log p(z(t_1)) = \log p(z(t_0)) - \int_{t_0}^{t_1} Tr\left(\frac{\partial f}{\partial z(t)}\right) dt.$$

This combination of NeuralODEs and Normalizing Flows is called *Continuous Normalizing Flows (CNFs)* and can be visualized on Figure 5.3. One huge difference in the continuous case, is that instead of computing the determinant of the Jacobian, we only need to calculate the trace. Determinants are generally calculated in $\mathcal{O}(d^3)$, however, the trace is a linear cost operation.

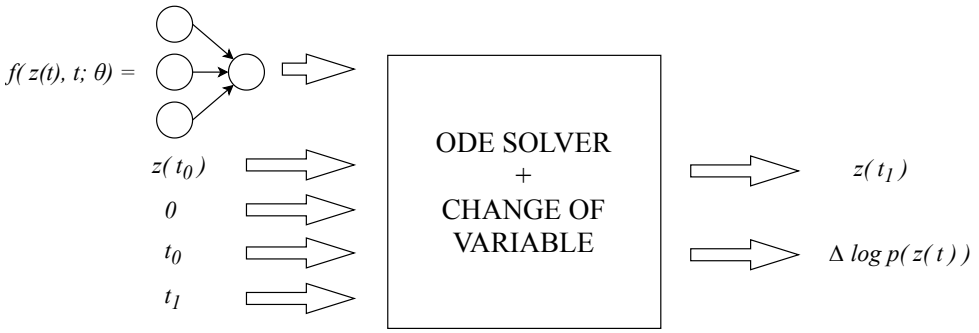

Figure 5.3: *$f(z(t),\ t;\ \theta)$ is neural network specifying the differential equation, $z(t_0)$ is the initial state, 0 is the initial log-density, t_0 is the initial time, t_1 is the final time, $z(t_1)$ is the final state and $\Delta \log p(z(t))$ is the change in log-density.*

Unfortunately, the overall complexity of the method above is still $\mathcal{O}(d^2)$ due to the Jacobian, which even though better, is still restrictive. Grathwohl et al. [8] further optimize the above method and reduce the overall complexity to

$\mathcal{O}(d)$. They achieve this in two steps. First, Vector-Jacobian products can be computed efficiently using reverse-mode automatic differentiation. Secondly, they show that they can get unbiased estimates of the trace of the Jacobian, by using the Hutchinson's trace estimator [15]. Consequently, they claim that Continuous Normalizing Flows implemented in this fashion can be considered unrestricted due to the free-form Jacobian of the transformation f.

Grathwohl et al. [8] additionally provide a PyTorch framework[1] with the previously mentioned improvements. Within their framework we can apply CNFs to random variables and log densities as

$$\underbrace{\begin{bmatrix} z(t_1) \\ \Delta \log p(z_t) \end{bmatrix}}_{Solution} = \int_{t_0}^{t_1} \underbrace{\begin{bmatrix} f(z(t), t; \theta) \\ Tr\left(\frac{\partial f}{\partial z(t)}\right) \end{bmatrix}}_{Dynamics} dt = cnf\left(\underbrace{\begin{bmatrix} z(t_0) \\ 0 \end{bmatrix}}_{InitialValues}, \; t_0, \; t_1, \; f \right)$$

and finally we can obtain $\log p(z_{t_1})$ as

$$\log p(z_{t_1}) = \log p(z_{t_0}) - \Delta \log p(z_t).$$

[1] https://github.com/rtqichen/ffjord/

Chapter 6

CNFs for Language Modelling

6.1 Introduction

It was previously shown how Normalizing Flows are a powerful technique to distort simple distributions into complex ones. Until recently, this was only feasible on small toy datasets, however, due to the advances in Chen et al. [3] and Grathwohl et al. [8] and moving to the continuous domain, it has become a viable technique for solving more complex problems.

This chapter goes through several different ways of incorporating Continuous Normalizing Flows in language modelling and discusses both their theoretical and practical implications.

6.2 Regression Over Word Embeddings

One possible approach is to take the final hidden state h from an RNN, and use it to parameterize a simple initial distribution P_h^0, and sample z_0 from it. Then, we can use CNFs to perform transformations on both z_0 and its log density in parallel. The goal is to transform z_0 into the embedding of the word we are trying to predict.

Let V be the vocabulary, h be the final hidden state, d be both the hidden state and the embedding dimensionality and $e_w^* \in R^d$ be the word embedding of the ground truth word. Let $P_h^0(\ .\ )$ be the probability density function of the initial distribution parameterized by the hidden state and f be a neural network architecture parameterizing the gradient as described in chapters 4 and 5. Then:

$$\begin{aligned} z_{t_0} &\sim P_0 \\ \log p(z_{t_0}) &= \log P(z_{t_0}; h) \\ z_{t_1}, \Delta \log p(z_t) &= cnf(z_{t_0}, \vec{0}, t_0, t_1, f) \\ \log p(z_{t_1}) &= \log p(z_{t_0}) - \Delta \log p(z_t) \end{aligned} \tag{6.1}$$

We can then treat the problem as a regression problem over word embeddings, i.e. we try to obtain a z_{t_1} as close as possible to e_w^* based on some distance metric. We can achieve that by assuming some distribution on the error and then train by minimizing the negative log-likelihood. One possible choice for such a distribution is the Von Mises-Fisher as proposed by Kumar and Tsvetkov [18].

Let e_w^* be the ground truth embedding and e_w be the predicted embedding. The density of the predicted embedding given the ground truth embedding is given by

$$p(e_w; e_w^*) = vMF(e_w; e_w^*) = C_m(\|e_w^*\|) e^{e_w^{*T} e_w},$$

where C_m is the normalization constant and is defined as

$$C_m(k) = \frac{k^{m/2-1}}{(2\pi)^{m/2} I_{m/2-1}(k)}.$$

I_v is a Bessel function of the first kind of order v and m is the dimensionality of the embeddings. The Negative Log-Likelihood then becomes

$$NLLvMF(e_w; e_w^*) = -\log C_m(||e_w||) - e_w^T e_w^*.$$

Consequently, by taking equation 6.1 into account the loss becomes

$$loss = NLLvMF(z_1;\ e_w^*).$$

Even though we do not explicitly use $\log p(z_{t_1})$ for training, at the end of this process we end up with a continuous probability distribution over the entire embedding space. To obtain a distribution over the vocabulary, we have to transform this continuous distribution into a discrete one. Additionally, if word embeddings are not fixed, we would have to add negative samples to prevent clustering them together.

One drawback of this approach is that this model is not explicitly trained to minimize perplexity. In this thesis, the focus is on models that explicitly minimize perplexity, which is why this type of models was not analyzed further.

6.3 CNF Language Models

6.3.1 Training LMs with Cross-Entropy

The de facto metric for evaluating language models is perplexity, which is why the majority of them are trained using Cross-Entropy. Recall that the Cross-Entropy between two distributions p and q is defined as

$$CE(p,\ q) = -\sum_{x} p(x) \log_b q(x)$$

and can be interpreted as the average amount of bits needed to encode the outcome of the distribution p based on a scheme optimized for distribution q. Perplexity on the other hand is defined as

$$PPL(p,\ q) = b^{CE(p,\ q)} = b^{-\sum_x p(x) \log_b q(x)},$$

where typical choices for the base b are 2 or e, simply because logarithms with these bases are easy to compute. It makes no difference which one we go for as long we are consistent across both formulas. In language modeling we usually take p to be the true distribution and q to be model distribution obtained from a Softmax layer. As $p(x)$ is a one-hot encoded ground truth vector, this boils down to

$$CE(p,\ q) = -\log q(x^*), \tag{6.2}$$

where x^* is the ground truth word. From equation 6.2 we can deduce that in language modeling, minimizing Cross-Entropy is equivalent to minimizing the negative log-likelihood of q. Additionally, as p and q are distributions over words, we are interested in the averaged per-word perplexity. This is a measure of how good our model is, because a per-word perplexity value of k, means that our model's predictive power is just as good as guessing the next word randomly between k words. Therefore perplexity is considered to be the best metric for evaluating LMs.

6.3.2 CNFs for Language Modelling

Chapter 2 introduced two limitations on standard language models. Let us first introduce how can one apply CNFs to language modeling and what limitations and benefits we get from using them.

Let V be the vocabulary, $h \in R^d$ be the final hidden state, $\log P_h^0(\ .\ )$ be the initial log-densities commonly obtained from a linear layer with weights

$E \in R^{|V| \times d}$ and additional biases. Finally, let f be a neural network parameterizing the gradient. Then, the forward pass for one training sample would look like:

$$z_{t_0} = E,\ z_{t_0} \in R^{|V| \times d} \tag{6.3}$$

$$\log pz_{t_0} = \log P_h^0(z_{t_0}),\ \log pz_{t_0} \in R^{|V|} \tag{6.4}$$

$$z_{t_1}, \Delta \log pz_t = cnf(z_{t_0},\ \vec{0},\ t_0,\ t_1,\ f) \tag{6.5}$$

$$\log pz_{t_1} = \log pz_{t_0} - \Delta \log pz_t \tag{6.6}$$

Let us first clarify what 6.3 and 6.4 mean. E are the weights of the final linear layer used to obtain the logits in standard LMs. A linear layer performs a matrix-vector product between the hidden state h and the weights E and adds biases to obtain a resulting vector of size equal to the vocabulary. The components of this vector will correspond to a shifted dot product between h and the corresponding row in E. The rows of E are commonly known as *output word embeddings*, contrary to the *input word embeddings* of the initial embedding layer. Additionally, if the weights between the input and the output layers are tied [16], E is also the embedding matrix. This means that given a hidden state, we obtain a distribution over the word embedding space, with the logits being the corresponding log-densities for each word. Therefore, we can treat the rows of E as the discrete set of values from this distribution, with log-densities equal to the logits.

Once we have an initial distribution, in equation 6.5 we obtain the change in log-density using CNFs with initial values $z_{t_0} \in R^{|V| \times d}$ and $\vec{0} \in R^{|V|}$. Furthermore, f is an arbitrary neural network and one possible way to define it is

$$\frac{\partial z(t)}{\partial t} = f(z(t),\ t) = W_2\ ReLU(W_1\ [z(t),\ t]^T)$$

where $W_1 \in R^{d\ \times\ d+1}$ and $W_2 \in R^{d \times d}$.

In equation 6.6 we obtain the final log-densities by subtracting the change in log-density from the initial distribution. At the end, we obtain log-densities of a distribution over the embedding space. To transform them to discrete probabilities, we can simply take the softmax:

$$q = Softmax(\log pz_{t_1})$$

Finally, we train with Cross-Entropy between the obtained model distribution q and the true distribution p represented by a one hot encoded ground truth vector:

$$loss = CrossEntropy(p, q)$$

As the CNF does not depend on the hidden state, regardless of how many samples we have in a single batch, we only need to perform one flow to obtain the change in log-density. This drastically reduces both the time and memory complexity, however it does mean that we end up with a transformation that is still limited by the *Softmax Bottleneck* problem. When the CNF does not depend on the hidden state it means that the change in log-density for every word is independent of the context. In return, this means that if we represent our model with a matrix similarly to the one in section 2.1 what we are going to get is

$$\begin{bmatrix} \log P_0(w_1|c_1) - \Delta \log P(w_1) & \dots & \log P_0(w_M|c_1) - \Delta \log P(w_M) \\ \vdots & \ddots & \vdots \\ \log P_0(w_1|c_N) - \Delta \log P(w_1) & \dots & \log P_0(w_M|c_N) - \Delta \log P(w_M) \end{bmatrix}$$

where $P_0(w_i|c_j)$ is the initial distribution for the i-th word given the j-th context and corresponds to $\log pz_{t_0}$ in equation 6.6. Additionally, $\Delta \log P(w_i)$ is the change in log-density and corresponds to $\Delta \log pz_t$ in equation 6.6. We can then split this matrix into two separate matrices A

$$A = \begin{bmatrix} \log P_0(w_1|c_1) & \dots & \log P_0(w_M|c_1) \\ \vdots & \ddots & \vdots \\ \log P_0(w_1|c_N) & \dots & \log P_0(w_M|c_N) \end{bmatrix}$$

and B

$$B = \begin{bmatrix} \Delta \log P(w_1) & \dots & \Delta \log P(w_M) \\ \vdots & \ddots & \vdots \\ \Delta \log P(w_1) & \dots & \Delta \log P(w_M) \end{bmatrix}.$$

Now the original matrix can be written as

$$A - B.$$

We know that the rank of the sum of two matrices is bounded by

$$\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B).$$

As it was previously shown that A is a low rank matrix and $\text{rank}(B) = 1$, we can conclude that this approach indeed does not solve the *Softmax Bottleneck* problem. Additionally, this model is also restricted by the *Single Transformation* problem described in section 2.2. In the next section, it is discussed how certain changes can release the model from both limitations.

6.4 Context Conditioned CNFs

Chapter 2 discusses the current limitations on language modeling. Moreover, in the previous section a novel approach that uses CNFs was introduced that still is still bounded by these limitations. In this section we go through a possible way to upgrade the previously described method in a way that it is not bounded by both the *Softmax Bottleneck* and the *Single Transformation* problems. Namely, the later denotes that we would like to adjust the nature of the distribution based on the context. We can do this by making f depend on h, i.e. f becomes

$$f(z(t),\ t,\ h)$$

and one way to define it is

$$f(z(t),\ t,\ h) = W_2\ ReLU(W_1\ [z(t),\ t]^T + W_h h), \tag{6.7}$$

where $W_1, W_h \in R^{d\ \times\ d+1}$ and $W_2 \in R^{d \times d}$.

Let V be the vocabulary, $h \in R^d$ be the final hidden state, $\log P_h^0(\ .\ )$ be the initial log-densities commonly obtained from a linear layer with weights $E \in R^{|V| \times d}$ (if the weights are tied [16] this is also the embedding matrix) and additional biases. Finally, let f be a neural network parameterizing the gradient as defined in equation 6.7. Then, the batched version of the forward pass for one training sample looks like

$$z_{t_0} = E,\ z_{t_0} \in R^{|V| \times d} \tag{6.8}$$

$$\log pz_{t_0} = \log P_h^0(z_{t_0}),\ \log pz_{t_0} \in R^{|V|} \tag{6.9}$$

$$z_{t_1}, \Delta \log pz_t = cnf(z_{t_0},\ \vec{0},\ h,\ t_0,\ t_1,\ f) \tag{6.10}$$

$$\log pz_{t_1} = \log pz_{t_0} - \Delta \log pz_t \tag{6.11}$$

Even though the approach is same as in section 6.3, here the CNF depends on h. We can then proceed and obtain the model distribution q with

$$q = Softmax(logpz_{t_1})$$

Finally, we can train the model with Cross-Entropy

$$loss = CrossEntropy(p, q)$$

where p is the true distribution and is represented by a one-hot encoded ground truth vector.

To see whether this model still suffers from the *Softmax Bottleneck* problem, we can do the same test as in the previous section. Let us first create a matrix for our model in the same manner

$$\begin{bmatrix} \log P_0(w_1|c_1) - \Delta \log P(w_1|c_1) & \dots & \log P_0(w_M|c_1) - \Delta \log P(w_M|c_1) \\ \vdots & \ddots & \vdots \\ \log P_0(w_1|c_N) - \Delta \log P(w_1|c_N) & \dots & \log P_0(w_M|c_N) - \Delta \log P(w_M|c_N) \end{bmatrix}$$

where $P_0(w_i|c_j)$ is the same initial distribution as in the previous case and corresponds to $\log pz_{t_0}$ in equation 6.11. Additionally, $\Delta \log P(w_i|c_j)$ is the change in log-density and corresponds to $\Delta \log pz_t$ in equation 6.11. Notice that in this case, the change in log-density depends on the context. Then, we can split this into two matrices, A

$$A = \begin{bmatrix} \log P_0(w_1|c_1) & \dots & \log P_0(w_M|c_1) \\ \vdots & \ddots & \vdots \\ \log P_0(w_1|c_N) & \dots & \log P_0(w_M|c_N) \end{bmatrix}$$

and B

$$B = \begin{bmatrix} \Delta \log P(w_1|c_1) & \dots & \Delta \log P(w_M|c_1) \\ \vdots & \ddots & \vdots \\ \Delta \log P(w_1|c_N) & \dots & \Delta \log P(w_M|c_N) \end{bmatrix}.$$

The rank of the original matrix is still bounded by the same rule, i.e.

$$\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$$

however, the rank of matrix B can now go up to N or M, i.e. $\text{rank}(B) \leq min(N,\ M)$. Therefore, the rank of $A + B$ is not constrained by the rank of A and the sum can potentially be a full-rank matrix. This proves that the model is indeed not limited by the *Softmax Bottleneck*. Additionally, this model starts with a simple initial distribution and distorts it into an arbitrary complex one based on the context. This means that the model is not limited by the *Single Transformation* problem by construction, as we adapt the nature of the distribution based on the context. Therefore, the entire model is more powerful and unrestricted. However, this slight change has certain implications on the computational efficiency of the entire method.

6.5 Issues with Context Conditioned CNFs

Having context conditioned CNFs drastically increases both the time and the memory complexity. Contrary to the previous case where regardless of how many samples in a batch we have solving one CNF was enough, now we need to solve a separate CNF for every distribution. During training, samples are typically represented as tensors of shape

$$[batch\_size,\ sequence\_size,\ embedding\_size],$$

where *sequence_size* stands for the amount of words in a single sample and *batch_size* stands for the amount of samples in a single batch. As for every sample we need to obtain *sequence_size* distributions, after we process the input with an LM we get a tensor of shape

$$[batch\_size,\ sequence\_size,\ vocabulary\_size],$$

which contains $batch\_size \times sequence\_size$ distributions over the vocabulary. This tensor in fact represents the initial distributions for the entire batch. For every distribution we now need to solve a CNF and distort it. If we batch the entire process and solve one large CNF, the tensor z_{t_0} from equation 6.8 has shape

$$[batch\_size,\ sequence\_size,\ vocabulary\_size,\ embedding\_size].$$

The size of the vocabulary in word-based LMs typically starts around 10^4 and can go up to 10^6 in datasets like the *The One Billion Word Benchmark* [2]. The dimensionality of the embeddings is typically in the low hundreds and reducing it past a certain point results in a loss of performance. Moreover, reducing *batch_size* and *sequence_size* generally only results in a memory versus speed trade-off. This means that evaluating the normalization constant is the real bottleneck for this model.

Possible way to avoid this vocabulary bottleneck is to use character-based LMs. Character based LMs model distributions over sequences of characters, i.e.

$$P(c_1, ..., c_n) = \prod_i^n P(c_i | c_{1..i-1}),$$

where c_i is the i-th character in the sequence. Regardless of how many words there are in the training corpus, the vocabulary of character-based LMs is

in most cases less than 100. Therefore, the vocabulary is never an issue for character based LMs. However modeling distributions over characters is more complicated than modeling a distribution over words. This follows from the fact that a character can exist in more contexts compared to a word, so modelling all of them is more complicated than in the case of word-based LMs.

However, if we really want to use word based LMs we would have to either resort to techniques like Hierarchical Softmax [24] or approximate the normalization constant using sampling techniques. Hierarchical Softmax comes with additional problems, such as choosing the ordering of the words, which is why in this thesis the focus is on approximating the Softmax via sampling.

6.6 Softmax Approximations

6.6.1 Introduction to Softmax Approximation

As shown in the previous section, evaluating the normalization constant for distributions over large vocabularies is not always possible. However, training large vocabulary LMs is still an active research area. *The One Billion Word Benchmark* [2] is a large vocabulary benchmark dataset where researchers push the limits and effectively train large vocabulary LMs. In this section we go through approaches that estimate the constant with only a few words.

6.6.2 Sampling Based Approaches

Equation 6.2 shows depicts a connection between Cross-Entropy and Negative Log-Likelihood. The Negative Log-Likelihood for a single word and context pair in language modelling can be written as

$$L_{w,c} = -\log \frac{\exp(l_{w,c})}{\sum_{w_i \in V} \exp(l_{w_i,c})},$$

where $l_{w,c}$ represents the logit for word w given context c. If we expand we get

$$L_{w,c} = -l_{w,c} + \log \sum_{w_i \in V} \exp(l_{w_i,c}).$$

Then, if we take the gradient with respect to the model's parameters θ we obtain

$$\nabla_\theta L_{w,c} = -\nabla_\theta l_{w,c} + \nabla_\theta \log \sum_{w_i \in V} \exp(l_{w_i,c}) \tag{6.12}$$

$$= -\nabla_\theta l_{w,c} + \frac{1}{\sum_{w_i \in V} \exp(l_{w_i,c})} \sum_{w_i \in V} \nabla_\theta \exp(l_{w_i,c}) \tag{6.13}$$

$$= -\nabla_\theta l_{w,c} + \frac{1}{\sum_{w_i \in V} \exp(l_{w_i,c})} \sum_{w_i \in V} \exp(l_{w_i,c}) \nabla_\theta l_{w_i,c} \tag{6.14}$$

$$= -\nabla_\theta l_{w,c} + \sum_{w_i \in V} \frac{\exp(l_{w_i,c})}{\sum_{w_j \in V} \exp(l_{w_j,c})} \nabla_\theta l_{w_i,c} \tag{6.15}$$

The fraction inside the sum is the model's probability of w_i given context c

$$P_\theta(w_i|c) = \frac{\exp(l_{w_i,c})}{\sum_{w_j \in V} \exp(l_{w_j,c})},$$

so the gradient can be written as

$$\nabla_\theta L_{w,c} = -\nabla_\theta l_{w,c} + \sum_{w_i \in V} P_\theta(w_i|c) \nabla_\theta l_{w_i,c} \tag{6.16}$$

$$= -\nabla_\theta l_{w,c} + \mathbb{E}_{w_i \sim P_\theta}[\nabla_\theta l_{w_i,c}] \tag{6.17}$$

Sampling based approaches estimate the expectation in equation 6.17.

6.6.3 Importance Sampling

We can approximate the expected value $\mathbb{E}$ of any probability distribution using Monte-Carlo methods. In our case, we can approximate the expected value in equation 6.17 with

$$\mathbb{E}_{w_i \sim P_\theta} \nabla_\theta l_{w_i,c} \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta l_{w_i,c},$$

where we obtain m samples from $P_\theta(w|c)$ and average the gradients. Unfortunately, $P_\theta(w|c)$ is what we are trying to learn so we have to use a propositional, or also called a noise, distribution $Q(w)$ from which it is cheap to sample as a substitute. Furthermore, it is important that Q is similar to P, which is why it is taken to be the unigram distribution of the training set in the case of langauge modelling. The goal of *Importance Sampling* is exactly that, to approximate a target distribution P using a propositional

distribution Q and Monte-Carlo methods. Namely, instead of weighting the gradient in equation 6.16 with the expensive to compute $P_\theta(w|c)$, we weight it with a factor that depends on $Q(w)$.

A standard practice is to obtain k noise samples from Q and estimate the aforementioned quantity with them. Noise Contrastive Estimation (NCE) [9] proposes using a surrogate loss and it models the problem as a binary classification task, where the goal is to predict whether a sample comes from the true or the noise distribution. Gutmann and Hyvärinen [9] show that as we increase the number of samples, the derivative of the NCE loss approaches the derivative of the softmax function.

Jozefowicz et al. [17] however, propose a different approach based on Importance Sampling. If we take the ground truth word and k noise samples and define

$$S = \{w_1, ..., w_{k+1}\},$$

such that w_1 is always the true word and $w_2, ...w_{k+1}$ are the noise samples we can optimize a multi-class loss over a multinomial variable Y representing the labels. Namely, we can define

$$P(Y = k \mid S,\ c) = Softmax(l_{w_k,c} - \log Q(w_k))$$

and train by maximizing the log-likelihood $\log P(Y = 1 \mid S,\ c)$. After training, $Softmax(l_{w,c})$ is a good approximation of $P_\theta(w|c)$. Jozefowicz et al. [17] suggest that this probably is a better choice than NCE for language modeling, as it optimizes a mutli-class classification task instead of a binary one.

Chapter 7

Experiments

7.1 Legend

This section explains what the abbreviations in the tables below mean.

- *model* - the model being used. AWD stands for AWD-LSTM [21], MoS [36] stands for Mixture of Softmaxes and DoC stands for Direct Output Connections [33].
- *exp* - number of experts. Models that perform a mixture of distributions need a prespecified value for the number of components in the mixture.
- *h* - dimensionality of the middle hidden states of the RNN.
- *lasth* - dimensionality of the final hidden state of the RNN.
- *emb* - dimensionality of the embeddings.
- *lr* - learning rate.
- *ep* - epoch at which the presented results are obtained.
- *vloss* / *tloss* - validation loss / test loss. Loss obtained on the validation or the test set.
- *vppl* / *tppl* - validation perplexity / test perplexity. Perplexity obtained on the validation or the test set.
- *vbpc* / *tbpc* - validation bits per character / test bits per character. Bits per character obtained on the validation or the test set.
- *prefinetuned* - in the case of transfer learning specifies whether the base model was finetuned before transferring the weights.
- *freeze* - in the case of transfer learning specifies whether the transfered weights are fixed or trainable.

Table 7.1: *Results from baseline word-based models before finetuning.*

model	exp	h	lasth	emb	lr	ep	vloss	vppl	tloss	tppl
AWD	n/a	960	400	400	20	517	4.11	60.93	4.07	58.67
MoS	15	960	620	280	20	511	4.06	57.89	4.02	55.84
DOC	15	960	620	280	20	500	4.02	55.45	3.98	53.44

Table 7.2: *Results from baseline word-based models after finetuning.*

model	vloss	vppl	tloss	tppl
AWD	4.10	60.33	4.06	58.05
MoS	4.04	56.73	4.00	54.54
DOC	4.00	54.68	3.97	52.87

7.2 Datasets

All models are evaluated on the Penn Treebank dataset which is the standard dataset for evaluating language models. The dataset is used as preprocessed by Mikolov et al. [23] and it consists of 929k training words, 73k validation words, and 82k test words. After preprocessing, all words consist of only lowercase letters and all numbers are replaced with a placeholder N. Additionally, newlines are replaced with a special <eos> token. Finally, after preprocessing the dataset contains only the 10k most frequent words and the rest are replaced with a special <unk> token.

7.3 Word Based Models

7.3.1 Baselines

The following three models were used as baselines

1. AWD-LSTM [21]
2. MoS [36]
3. DOC [33]

For every baseline, the latest hyperparameters proposed in their corresponding github repositories were used. Unfortunately, due to changes in PyTorch versions, exact reproduction of their results was not possible. The results before finetuning are presented in table 7.1 and the results after finetuning are presented in table 7.2.

Table 7.3: *Results from performing NeuralODE-based transformations on top of the logits of a pre-trained AWD-LSTM model. Several different experiments manage to improve on the baseline. The most notable is experiment 1, as it improves on the baseline by a whole perplexity point. The perplexities on the validation and test sets for this experiment can be seen in bold. Additionally, for this particular model using a learning rate greater or equal to 0.1 results in instant overfitting.*

#	prefinetuned	freeze	lr	ep	vloss	vppl	tloss	tppl
1	no	no	0.01	12	4.09	**59.94**	4.06	**57.71**
2	yes	no	0.01	5	4.10	60.50	4.06	58.09
3	no	yes	0.01	4	4.11	60.73	4.07	58.68
4	yes	yes	0.01	4	4.10	60.56	4.06	58.25
5	no	no	0.1	1	4.09	60.02	4.06	57.73
6	yes	no	0.1	1	4.11	60.79	4.06	58.25
7	no	yes	0.1	1	4.11	60.80	4.07	58.74
8	yes	yes	0.1	1	4.11	60.65	4.07	58.33

7.3.2 NeuralODE Logit Transformations

This model is based on Neural ODEs and is explained in section 4.3. It performs nonlinear transformations on the logits by using an ODENet on top of the initial AWD-LSTM model. The architecture of the neural network, that specifies the differential equation, in the experiments is defined as

$$f(l,\ t) = W\ Softplus(W_f^T \begin{bmatrix} l \\ t \end{bmatrix}),\ W_f \in R^{|V|+1\ \times\ k},\ W \in R^{|V|\ \times\ k},$$

where l are the logits obtained from a pre-trained AWD-LSTM, t is the time and W, W_f are trainable parameters. Additionally, $|V|$ is the size of the vocabulary and k is a dimensionality bottleneck and set to be equal to the dimensionality of the embeddings. The hyperparameters being used are the latest hyperparameters proposed in the official AWD-LSTM repository. The results of the experiments for this model are presented in table 7.3.

7.3.3 Continuous Normalizing Flows

Basic CNFs

This is the most simple model based on Continuous Normalizing Flows and is explained in section 6.3. It is more efficient in terms of speed, however theoretically less expressive than the Context Conditioned CNFs model, as it still suffers from both limitations mentioned in chapter 2. Nonetheless, using the flexibility of CNFs in addition to the base models, seem to be beneficial as the model manages to improve on some of the baselines.

Table 7.4: *Results from training CNFs on top of a pre-trained AWD-LSTM word-based model. Experiment 3 improves on the baseline by 0.3 perplexity points. The perplexities on the validation and test sets for this experiment, can be seen in bold.*

#	prefinetuned	freeze	lr	ep	vloss	vppl	tloss	tppl
1	no	no	0.1	152	4.12	61.56	4.07	58.80
2	yes	no	0.1	160	4.15	63.20	4.11	60.97
3	no	yes	0.1	80	4.11	**60.65**	4.07	**58.53**
4	yes	yes	0.1	44	4.10	60.51	4.06	58.21
5	no	no	1	73	410	60.58	4.06	57.88
6	yes	no	1	43	4.11	60.98	4.07	58.28
7	no	yes	1	27	4.11	61.06	4.07	58.75
8	yes	yes	1	21	4.12	61.31	4.08	58.88

In the experiments, the neural network's architecture, that specifies the differential equation is defined as

$$f(l,\ t) = W\ Softplus(W_f^T \begin{bmatrix} l \\ t \end{bmatrix}),\ W_f \in R^{d+1\ \times\ d},\ W \in R^{d\ \times\ d},$$

where d is the dimensionality of the embeddings. The hyperparameters being used are the latest hyperparameters proposed in the corresponding repositories of the baseline models.

The results of using CNFs on top of a pre-trained AWD-LSTM model can be seen in table 7.4. The results of using CNFs on top of a pre-trained MoS model can be seen in table 7.5. Finally, the results of using CNFs on top of a pre-trained DoC model can be seen in table 7.6.

Context Conditioned CNFs

Due to the issues discussed in section 6.5 all word-based Context Conditioned CNFs are trained using Importance Sampling. In every training iteration, 20 labels are obtained from the unigram distribution of the training set and are concatenated to the true label. This drastically speeds up training, however evaluating on the original validation and test sets remains unfeasible. Therefore, when evaluating Context Conditioned CNFs, only the first 400 samples from the validation and the test sets are used.

The hyperparameters being used are the latest hyperparameters proposed in the official MoS repository. Furthermore, the RNN base of the model is initialized with the weights of a pre-trained AWD-LSTM model. Additionally, the architecture of the neural network, that specifies the differential equation, in the experiments is defined as

Table 7.5: *Results from training CNFs on top of a pre-trained MoS word-based model. Experiments 3, 4 and 8 manage to improve on the baseline. Their perplexities on the test and validation set can be seen in bold. For this particular model, freezing the transferred weights of the MoS model and only training the CNF weights results in lower perplexity values.*

#	prefinetuned	freeze	lr	ep	vloss	vppl	tloss	tppl
1	no	no	0.01	61	4.07	58.62	4.03	56.45
2	yes	no	0.01	58	4.06	57.80	4.02	55.46
3	no	yes	0.01	355	4.06	**57.70**	4.02	**55.78**
4	yes	yes	0.01	338	4.04	**56.65**	4.00	**54.53**
5	no	no	0.1	7	4.09	59.49	4.05	57.29
6	yes	no	0.1	7	4.07	58.77	4.03	56.35
7	no	yes	0.1	62	4.05	57.56	4.02	55.68
8	yes	yes	0.1	62	4.03	**56.50**	4.00	**54.42**

Table 7.6: *Results from training CNFs on top of a pre-trained DoC word-based model. None of the experiments improve on the baselines. Similarly to previous experiments, smaller learning rates and freezing the transferred weights of the base model achieves lower perplexity values.*

#	prefinetuned	freeze	lr	ep	vloss	vppl	tloss	tppl
1	no	no	0.05	11	4.04	56.69	4.00	54.54
2	yes	no	0.05	8	4.03	56.05	3.99	54.15
3	no	yes	0.05	81	4.02	55.55	3.98	53.59
4	yes	yes	0.05	64	4.00	54.79	3.97	53.02
5	no	no	0.1	11	4.05	57.16	4.00	54.85
6	no	yes	0.1	43	4.02	55.63	3.98	53.66
7	yes	yes	0.1	46	4.00	54.86	3.97	53.14
8	no	no	1	49	4.06	57.84	4.01	55.14
9	no	yes	1	11	4.07	58.62	4.04	56.58
10	yes	yes	1	15	4.06	58.00	4.03	56.10

Table 7.7: *Results from training Context Conditioned CNFs on top of a pre-trained word-based AWD-LSTM model. The perplexities are computed only on the first 400 samples of the validation and test set. No experiment improves on the baseline in table 7.8. However, the baseline model was trained by evaluating the full partition function of the Softmax, and this model is trained by randomly sampling 20 labels from the unigram distribution of the training set in every iteration. Due to the small number of labels in every iteration and the randomness, a lot more epochs are needed to reach or improve on the performance of the baseline model.*

#	prefinetuned	freeze	lr	ep	vloss	vppl	tloss	tppl
1	no	no	0.01	84	4.12	61.45	3.93	51.09
2	yes	no	0.01	84	4.10	60.52	3.96	52.48
3	no	yes	0.01	46	4.12	61.72	3.94	51.35
4	yes	yes	0.01	46	4.10	60.14	3.96	52.58
5	no	no	0.1	52	4.11	61.15	3.94	51.51
6	yes	no	0.1	45	4.10	60.64	3.96	52.48
7	no	yes	0.1	24	4.13	62.22	3.95	52.14
8	yes	yes	0.1	24	4.11	60.85	3.95	52.00

Table 7.8: *Perplexities of the baseline word-based AWD-LSTM model computed on the first 400 samples of the validation and test sets.*

model	vloss	vppl	tloss	tppl
AWD	4.09	59.64	3.92	50.30

$$
\begin{aligned}
f(l,\ t,\ h) &= W\ Softplus(W_f^T \begin{bmatrix} l \\ t \end{bmatrix} + W_h h) \\
W_f &\in R^{d+1\ \times\ d} \\
W, W_h &\in R^{d\ \times\ d},
\end{aligned}
$$

where l are the logits, t is time, h is the final hidden state obtained from the AWD-LSTM base and W, W_f and W_h are trainable parameters. The results can be seen in table 7.7. Additionally, for fair comparison, table 7.8 contains metrics for the AWD-LSTM model, computed on the first 400 samples of the validation and test sets.

7.4 Character Based Models

Training character-based models on Penn Tree Bank Mikolov et al. [23] is performed using same pre-processing as when training word-based models. This means that only the 10000 most frequent words are kept, and all others

Table 7.9: *Results from training baseline character-based models.*

model	h	lasth	emb	lr	ep	vloss	vbpc	tloss	tbpc
awd	1000	200	200	0.002	364	0.84	1.212	0.82	1.183

Table 7.10: *Results from training Context Conditioned CNFs on top of a pre-trained character-based AWD-LSTM model. Most of the experiments reach same performance as the baseline model. An issue with training Context Conditioned CNF's with transferred weights is the possibility of initializing the model in a local minimum. This means that the CNF learns weights almost equal to 0, in order to retrieve the initial distribution from the baseline model.*

#	freeze	lr	ep	vloss	vbpc	tloss	tbpc
1	yes	1e-5	55	0.84	1.213	0.82	1.184
2	no	1e-4	15	0.84	1.212	0.82	1.182
3	yes	1e-4	28	0.84	1.212	0.82	1.183
4	no	2e-3	1	0.92	1.329	0.90	1.295
5	yes	2e-3	3	0.84	1.212	0.82	1.183

are substituted with an <unk> token. This drastically reduces the number of possible transitions between characters and simplifies the problem.

Additionally, when using Context Conditioned CNFs for character models the size of the vocabulary is usually less than 50. Therefore, character based Context Conditioned CNFs do not suffer from the issues mentioned in section 6.5 and are trained using the entire vocabulary.

For character-based models only the AWD-LSTM [21] mode was used as a baseline. Similarly to the word-based models, the latest hyperparameters proposed in the official github repository were used. Exact reproduction of the results was not possible due to differences in PyTorch versions. The baseline results can be seen in table 7.9.

Evaluating the full partition function, even though feasible, is still not fast enough to train these models from scratch. Therefore, similarly to the case of word-based models, the RNN weights are initialized with the weights of the pre-trained baseline AWD-LSTM model. The results can be seen in table 7.10.

Chapter 8

Conclusion

Language modelling remains one of the most important subfields of Natural Language Processing. Additionally, its real life applications are increasing everyday. As such, it is an extremely active research area of extreme importance to the Natural Language Processing community. Especially important class of models are the RNN-based language models. Due to the relatively small number of parameters in comparison to transformer-based models, they represent the de facto model for commercial language modelling. However, recent findings [36] show several limitations on standard RNN-based language modelling.

In this thesis, I have introduced several different models for language modelling based on a novel concept. Namely, I have successfully integrated NeuralODEs [3] and Continuous Normalizing Flows [8] with RNN-based language models. For every model, I have introduced and discussed the theoretical and practical limitations. Finally, I have summarized my findings and I have managed to improve on some of the baselines.

8.1 Future Work

The main limitation of this novel family of language models is their computational complexity. Since fast and efficient training was not possible, there was no room for extensive hyperparameter search. This means that most of the models manage to either beat their respective baseline or get close to it, by using hyperparameters that are not optimized for them. As performing an extensive hyperparameter search usually results in an improvement of several metric points, it is interesting to see how much of an improvement we are going to see.

For the same reason, all models are trained by first initializing them with the weights of the baseline model they are compared against. It is possible that

this initialization puts the model in a sub-optimal region of the optimizational space, as seems to be the case with character-based models. Given enough resources and time, it is interesting to see how will these models perform when trained from scratch.

Appendix A

Abbreviations

NLP	Natural Language Processing
LM	Language Modelling
RNN	Reccurent Neural Network
LSTM	Long Short-Term Memory
MoS	Mixture of Softmaxes
DOC	Direct Output Connections
SGD	Stochastic Gradient Descent
ASGD	Averaged Stochastic Gradient Descent
NT-ASGD	Non-monotonically Triggered ASGD
ResNet	Residual Network
ODE	Ordinary Differential Equation
CNF	Continuous Normalizing Flow
NCE	Noise Contrastive Estimation

Bibliography

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[2] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[3] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] L. Dinh, D. Krueger, and Y. Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[6] Y. Gal and Z. Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

[7] O.-E. Ganea, S. Gelly, G. Bécigneul, and A. Severyn. Breaking the softmax bottleneck via learnable monotonic pointwise non-linearities. *arXiv preprint arXiv:1902.08077*, 2019.

[8] W. Grathwohl, R. T. Chen, J. Betterncourt, I. Sutskever, and D. Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

[9] M. Gutmann and A. Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings*

of the Thirteenth International Conference on Artificial Intelligence and Statistics, pages 297–304, 2010.

[10] D. Ha, A. Dai, and Q. V. Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[11] E. Haber and L. Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2017.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450, 1990.

[16] H. Inan, K. Khosravi, and R. Socher. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*, 2016.

[17] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[18] S. Kumar and Y. Tsvetkov. Von mises-fisher loss for training sequence to sequence models with continuous outputs. *arXiv preprint arXiv:1812.04616*, 2018.

[19] Y. Lu, A. Zhong, Q. Li, and B. Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *arXiv preprint arXiv:1710.10121*, 2017.

[20] J. H. Martin and D. Jurafsky. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall Upper Saddle River, 2009.

[21] S. Merity, N. S. Keskar, and R. Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[22] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[23] T. Mikolov, A. Deoras, S. Kombrink, L. Burget, and J. Černockỳ. Empirical evaluation and combination of advanced language modeling techniques. In *Twelfth Annual Conference of the International Speech Communication Association*, 2011.

[24] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[25] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2, 2012.

[26] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[27] L. S. Pontryagin. *Mathematical theory of optimal processes*. Routledge, 2018.

[28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1 (8), 2019.

[29] D. J. Rezende and S. Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[30] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[31] L. Ruthotto and E. Haber. Deep neural networks motivated by partial differential equations. *arXiv preprint arXiv:1804.04272*, 2018.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[33] S. Takase, J. Suzuki, and M. Nagata. Direct output connection for a high-rank language model. *arXiv preprint arXiv:1808.10143*, 2018.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[35] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[36] Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen. Breaking the softmax bottleneck: A high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017.