NEURAL ORDINARY DIFFERENTIAL EQUATIONS BASED LANGUAGE MODELLING

COMP5329 ASSIGNMENT 2 RESEARCH TRACK

Hongbo Du
500635346
University of Sydney
hodu9417@uni.sydney.edu.au

Yuhao Hu
470139936
University of Sydney
yuhu5454@uni.sydney.edu.au

Liam Watts
510562348
University of Sydney
lwat0536@uni.sydney.edu.au

May 29, 2021

ABSTRACT

We studied how Neural ODEs based models perform in language modeling field by reconstructing a new ODE-LSTM model and proposing an ODE-GPT-2 model by using the existing Neural ODE framework and other language modeling frameworks. Our model architectures involve the classical state-of-the-art models as encoder and outputs the generated words by Neural ODEs model. We also compared the performances between our ODE-based language models and the classical state-of-the-art language models such as LSTM and GPT-2.

***K*eywords** Neural Ordinary Differential Equations (NODE), Adjoint Method, LSTM, GPT-2, ODE-LSTM, ODE-GPT2

1 Introduction

We propose to investigate an interesting new area in Artificial Intelligence (AI) research. Specifically, we are interested in examining the applications of Neural Ordinary Differential Equations (ODE) in Natural Language Processing (NLP). The idea of Neural ODEs, a new family in deep neural networks, was introduced in 2018 by [Chen et al., 2018]. The existing research on Neural ODEs has shown great progress but cutting-edge techniques have shown that using non-neuron based ideas gives great performance improvement, such as Transformers which use self-attention to give state of the art performance on NLP and recently Computer Vision tasks. Our team wishes to further examine a less explored non-neuron based model, the ODE network, which represents a neural network as a differential equation. Unlike the recent breakthrough as Generative Adversarial Networks (GANs) [Goodfellow et al., 2014], the applications of Neural ODEs to real world problems have not been explored, especially in the NLP field. Hence, the aim of this study is to explore the novelty of Neural ODEs in NLP.

Current AI research is very focused on Neural Networks and Deep Neural Networks, all revolving around the concept of a neuron, a concept loosely based on neurons in the human brain. The classical neural networks are built up by a stack of discrete hidden layers, where the number of hidden layers are required to be specified initially, which represents the depth of this neural network. In contrast, Neural ODEs is an approach where there is no need to explicitly define the number of hidden layers. [Chen et al., 2018] suggest that the network is reconstructed as a continuous-depth model, whose output is computed by an ODE solver. The importance of such an approach is that it can reframe some traditional discrete-depth neural network architecture into a continuous-depth network. For example, the Residual Network (ResNet) yields its hidden states by $\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t + \theta_t)$, where $t \in \{0, \ldots, T\}$ and $\mathbf{h}_t \in \mathbb{R}^D$ [He et al., 2015]. Whereas the hidden layers in the ResNet can be parameterized by an ODE: $d\mathbf{h}(t)/dt = f(\mathbf{h}(t), \theta)$ [Chen et al., 2018]. Therefore, Neural ODEs can generalize the ResNet to continuous-depth models in most cases. This is one example to show the importance of continuous-depth neural network in state-of-the-art model. We would like to investigate whether there is a significant impact of using Neural ODEs in the place of traditional models in NLP.

We propose to build different Neural ODE models in python based on the existing ODE models. In particular, we will attempt to apply Neural ODE models to the NLP Language modelling task. We build Neural ODE based language modeling that uses LSTM and GPT-2 model as sequence encoder, then using the Neural ODE model as decoder to

output the generated sequence. The main motivation of using Neural ODE's for such a task is their unique ability to perform inference with constant memory usage. The current state of the art transformer language models must store intermediate hidden states for each sequence output, which requires a large amount of memory due to the size of each hidden state and the number of layers. In contrast as a Neural ODE model's output is calculated by solving a differential equation no intermediate hidden states need to be stored, the only information needed is the initial hidden state and derivative function describing how it transforms. In this study we aim to compare the performance between the classical state-of-the-art models and our ODE-based models on language modeling tasks, to investigate if Neural ODEs can be applied in language modeling.

2 Related Works

2.1 Neural ODE

[Chen et al., 2018] investigate the use of black-box ODE solvers as a model component for training a neural network. Unlike the discrete neural network, an ODE network defines a vector field that continuously transforms the state. For example, Neural ODEs can be considered as a continuous-depth ResNet. Consider that in a discrete-depth neural network, the hidden state is computed by

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t)$$

where t denotes the layer and $\mathbf{h}_t \in \mathbb{R}^d$ denotes the hidden state at layer t and $f(\cdot) : \mathbb{R}^d \to \mathbb{R}^d$ is some differentiable function that preserves the dimension of $\mathbf{h}_t$, according to [Dupont et al., 2019]. Consider that we think this formula in terms of initial value problem, we are given $\mathbf{h}(t_0)$, and we are trying to find

$$\mathbf{h}(t_1) = \mathbf{h}(t_0) + \int_{t_0}^{t_1} f(\mathbf{h}_t, \theta_t)dt$$

where $\frac{d\mathbf{h}}{dt} = f(\mathbf{h}_t, \theta_t)$ is the time derivative. [Shen et al., 2020] state that Euler method is extremely popular for solving the initial value problems. Note that the hidden state for ResNet is the same for the Euler method for solving ordinary differential equations. In particular,

$$\mathbf{h}_{t+1} = \mathbf{h}_t + hf(\mathbf{h}_t, \theta_t) \qquad \text{where } h = 1.$$

In the case of adding more layers, [Chen et al., 2018] propose that we can parameterize the continuous dynamics of hidden states using an ODE, which is computed by

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta).$$

In addition, [Dupont et al., 2019] let $\mathbf{h}(0) = \mathbf{x}$ denote the input layer at time 0, we can define the output layer $\mathbf{h}(T)$ to be the features learned by the model at time T. Finally, map an input $\mathbf{x}$ to an output $\mathbf{y}$ by solving an ODE starting from $\mathbf{x}$, which equivalents to solve the initial value problem to time T. [Chen et al., 2018] compute the value of $d\mathbf{h}(t)/dt$ by a black-box ODE solver to evaluate the hidden unit dynamics f. In general, the idea is to generalize the discrete-depth network to continuous-depth network, and training it by using an ODE solver. This is the fundamental idea of our neural network architecture, therefore it is essential to understand.

2.2 Augmented Neural ODE

One limitation of Neural ODE is that there are some functions that a Neural ODE cannot represent. [Dupont et al., 2019] introduce classes of functions in arbitrary dimension d. One such function is: let $0 < r_1 < r_2 < r_3$ and let $g : \mathbb{R}^2 \to \mathbb{R}$ be a function such that

$$\begin{cases} g(\mathbf{x}) = -1 & \text{if } ||\mathbf{x}|| \leq r_1 \\ g(\mathbf{x}) = 1 & \text{if } r_2 \leq ||\mathbf{x}|| \leq r_3 \end{cases}$$

where $||\cdot||$ is the Euclidean norm. This is a ring shaped feature with width $r_3 - r_2$, denoted as A, and a circle feature with radius of r_1 inside the ring, denoted as B. The goal is to linearly separate the two features such that $\phi(A) = 1$ and $\phi(B) = -1$, respectively, where ϕ is the feature mapping. [Dupont et al., 2019] stated that since B is inside of A so it is not possible to have linearly separable features mapping without crossing the trajectories. [Dupont et al., 2019] provide the rigorous mathematical proof that involving topology and ODE theory in their appendix. To overcome the limitation of intersecting trajectories, [Dupont et al., 2019] propose the idea of Augmented Neural ODEs that augments

the space from $\mathbb{R}$ to $\mathbb{R}^{d+p}$ from which we learn and solve the ODE. In particular, letting $\mathbf{a}(t) \in \mathbb{R}^p$ denote a point in the augmented space, the augmented ODE can be formulated as

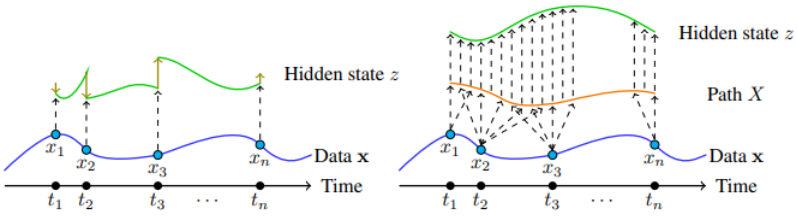
$$\frac{d}{dt}\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix} = \mathbf{f}\left(\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix}, t\right), \qquad \begin{bmatrix}\mathbf{h}(0)\\ \mathbf{a}(0)\end{bmatrix} = \begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix} \tag{1}$$

Essentially, the idea of the formula is to concatenate the data point $\mathbf{x}$ with a zero vector $\mathbf{0}$ then solve the ODE on this augmented space $\mathbb{R}^{p+d}$. In this way, ANODEs augment the space on which the ODE is solved, allowing the model to use the additional dimensions to learn more complex functions using simpler flows. For instance, the function above takes more steps and crossing trajectories in a 2-dimensional space, but ANODEs allow it to take fewer steps without crossing trajectories in a 3-dimensional space.

Based on their experiments, [Dupont et al., 2019] claim that this will make the augmented $\mathbf{f}$ smoother, giving rise to simpler flows that the ODE solver can compute in fewer steps. This results in achieving lower losses, better generalization and lower computational cost than regular Neural ODEs. However, the limitation for our research is that the augmented space may not be useful for language modelling as our input hidden states already have a large number of dimensions. The idea of ANODE, though, gives a simple extension on the fundamental Neural ODEs that generally gives improved performance.

2.3 Neural CDE

Neural Controlled Differential Equation (Neural CDE) is an idea proposed by [Kidger et al., 2020] to extend the Neural ODE model.



This demonstrates the difference between Neural ODEs (left) and Neural CDEs (right) [Kidger et al., 2020]

The hidden state of the Neural CDE model has continuous dependence on the observed data, whereas the hidden states of Neural ODEs are modified at each observation in terms of time series. To simplify the idea of Neural CDE, Neural ODEs are the continuous analogue of a ResNet as we discussed above and Neural CDEs are the continuous analogue of an RNN. [Kidger et al., 2020] state that Neural CDE is capable of processing partially observed input data, therefore, this is important to our research as our model is language generating model. The input data is considered as partially observed data. The mathematical architecture is defined as followings: let observations $x_i \in \mathbb{R}^v$ with timestamp $t_i \in \mathbb{R}$; let $X : [t_0, t_n] \to \mathbb{R}^{v+1}$ be a continuous function with bounded variation such that $X_{t_i} = (x_i, t_i)$, in other words, $\boldsymbol{x}$ is joined up to create the interpolation X; let $\zeta_\theta : \mathbb{R}^{v+1} \to \mathbb{R}^w$ be any neural network model depending on θ where w is the hidden state size; let $f_\theta : \mathbb{R}^w \to \mathbb{R}^{v+1}$ be any neural network model depending on θ;

$$z_t = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s) dX_s \qquad \text{for } t \in (t_0, t_n],$$

where $z_{t_0} = \zeta_\theta(x_0, t_0)$ which we have the initial condition depending on the first element of the time series. If we take the derivative of the above function, we obtain

$$\frac{dz_t}{dt} = f_\theta(z_t)\frac{dX_t}{dt}.$$

In particular, the left hand side means z is modified continuously according to this CDE, and the right hand side is a matrix-vector product between f_θ and dX_t/dt. In this way, the authors claims that the Neural CDE is naturally adapting to incoming data, since changes in X results in changes in z, that is, changes in X changes the local dynamics of the system. Notice that the equation above is still an ODE, therefore [Kidger et al., 2020] use adjoint method to train the Neural CDE as well. To simplify the idea of Neural CDE, Neural ODEs are the continuous analogue of a ResNet as we discussed above, the Neural CDE is the continuous analogue of an RNN. [Kidger et al., 2020] state that Neural CDE is capable of processing partially observed and irregularly sampled input data. This would be helpful for our research problem as the data process X can take in new data at each point without changing the architecture of the model.

2.4 ODE-LSTM

ODE-LSTM is a model proposed by [Lechner and Hasani, 2020]. ODE-LSTM aims to be an analogue to standard RNNs but represent them with ordinary differential equation, which uses ODE Solver to compute output over a continuous space. [Lechner and Hasani, 2020] state that the LSTMs express their discrete states as a pair (c_t, h_t), where c_t is the memory cell which ensures a constant error propagation; and h_t is the output state which enables the LSTM to learn non-linear dynamics. [Habiba and Pearlmutter, 2020] propose i_t is the input gate, f_t is the forget gate, o_t is the output gate, c_t is the cell update, and h_t is the output state. Those are calculated the same as the traditional LSTM update functions:

$$
\begin{aligned}
i_t &= \sigma(x_t W_{xi} + h_{t-1} W_{hi} + w_{ci} \odot c_{t-1} + b_i) \\
f_t &= \sigma(x_t W_{xf} + h_{t-1} W_{hf} + w_{cf} \odot c_{t-1} + b_f) \\
o_t &= \sigma(x_t W_{xo} + h_{t-1} W_{ho} + w_{co} \odot c_{t-1} + b_o) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \sigma(x_t W_{xc} + h_{t-1} W_{hc} + b_c) \\
h_t &= o_t \odot \sigma(c_t)
\end{aligned}
$$

where x_t is the input sequence, W_{ij}, b_{ij} for $i, j \in \{i, f, o, x, c, h\}$ are the weights and bias with respect to proper dimensions and σ is the sigmoid function. [Habiba and Pearlmutter, 2020] claim that ODE-LSTM pass the changes in input, forget, hidden gate to the ODE solver and uses only the changes in smaller fraction to compute the output, memory cell, and hidden state as:

$$
\begin{aligned}
\tilde{o_t} &= \frac{do_t}{dt} \\
c_t &= \frac{df_t}{dt} * c_0 + \frac{di_t}{dt} * c_t \\
h_t &= \tilde{o_t} * \frac{dc_t}{dt}
\end{aligned}
$$

Moreover, [Lechner and Hasani, 2020] describe the ODE-LSTM as $\mathrm{f}_\theta(x_{t+T}, (c_t, h_t), T) \rightarrow (c_{t+T}, h_{t+T})$, and assume the all weights and bias are initialized close to 0. Then c_t do not suffer from a vanishing or exploding gradient in the training process, in particular, $\frac{dc_{t+1}}{dc_t} = \sigma(a) \approx 0.73$. Therefore, the key advantage of ODE-LSTM is that it has a memory cell with forget gates controlled as we can increase the bias, which allows the model to learn long-term dependencies in the data. This feature might be helpful for us as long term dependency might happen in our task. However, one possible limitation is the memory cell's initialization.

2.5 Neural ODEs for Machine Translation

[Alba and Stronov, 2019] conduct experiment to show that Neural ODEs systems perform at least on par with standard LSTM-based NMT systems within the context of machine translation. They use the adjoint method to compute the gradients of loss function in the same fashion as the original Neural ODEs paper by [Chen et al., 2018]. The goal of this paper is to adapt this concept to machine translation tasks. [Alba and Stronov, 2019] pass the character embeddings into a 1D convolutional layer with max-pooling and ReLU activation. Then input the output of the convolutional layer $x_{convout}$ into ODEBlock to output the tuple $(x_{convout}, \text{ODESolver}(\mathcal{F}, x_{convout}, t_{int}))$ where $\mathcal{F}$ is the ODE function and t_{int} is the integration time used in the ODESolver. The output of the ODEBlock $\text{ODESolver}(\mathcal{F}, x_{convout}, t_{int})$ goes through the dropout layer and outputs the final word embeddings subsequently used in a Seq2Seq translation network. [Alba and Stronov, 2019] claim that they can decide whether to use the adjoint method for the ODESolver to train the neural network, and the adjoint method is used to increase the performance. The differential function $\mathcal{F}$ is the forward function which is implemented in various ways to compare the results. $\mathcal{F}$ takes t and x as input where t is a float value and x is $x_{convout}$. [Alba and Stronov, 2019] declare that the forward function NODE-t performs the best, which is implemented as

$$
\begin{aligned}
X^* &\leftarrow ReLU(x) \\
A^* &\leftarrow ReLU(W_t^{(1)} T + b_t^{(1)}) \qquad \text{where } W_t^{(1)} \in \mathbb{R}^{e_{word}+1 \times e_{word}}, b_t^{(1)} \in \mathbb{R}^{e_{word}} \\
\mathcal{F}_{t-concat}(t, x) &= x + W_t^{(2)} \tilde{T} + b_t^{(2)} \qquad \text{where } W_t^{(2)} \in \mathbb{R}^{e_{word}+1 \times e_{word}}, b_t^{(2)} \in \mathbb{R}^{e_{word}} \\
&\text{where } T = [\boldsymbol{t}|X^*] \text{ and } \tilde{T} = [\boldsymbol{t}|A^*] \text{ are the } t\text{-concatenated matrix}
\end{aligned}
$$

[Alba and Stronov, 2019] found that, for the task of machine translation, Neural ODE based model display significantly inefficiency in training/testing speed compared to trational NMT model. This paper is very helpful for our research

since it shows the feasibility of our goal, and the expected training speed for the ODE-based language model. Even the performance of ODE is similar to traditional NMT with the best forward function in ODESolver. This paper also adapt the Neural ODE concept to the language modeling, and it shows us how [Alba and Stronov, 2019] pass the embedding inputs into the ODEBlock.

3 Techniques

3.1 Dataset and preprocessing

Firstly, we load the word embeddings, created by [Pennington et al., 2014], `glove-wiki-gigaword-300` which has 400,000 vectors of 252MB file size. The dataset we use in this research is the Penn Treebank dataset [Marcus et al., 1993] in PyTorch. According to the description, the Penn Treebank contains one million words, in raw text, of 1989 Wall Street Journal (WSJ) which is of 98,732 stories for syntactic annotation. We pad sentences and convert words to their glove vector to get input features. Then we convert to one hot vocab and shift 1 to the left to get output labels. The reason is that converting to 1 hot takes too much memory, so just store indices and convert later. We use left padding, as we want the hidden state at the end (right) to ignore the padding. Once we finish the preprocessing, we obtain `word_vector_dataset` and `labels`.

Secondly, aside from the Penn TreeBank, we decided to use WikiText-2 [Merity et al., 2017] as well to get a better look at the performance. This WikiText language modeling dataset is a collection of over 100 million word tokens that are extracted from the certified articles on Wikipedia. Compared to the Penn Treebank we mentioned above, WikiText-2 is over 2 times larger, with a more modern context. The WikiText-2 also features a far larger vocabulary and retains the original case, punctuation and numbers - all of which are removed in Penn Treebank. Since it is composed of full articles, the dataset is well suited for models that can take advantage of relatively slong term dependencies.

3.2 Adjoint method

The adjoint sensitivity method is an approach that introduced by [Pontryagin et al., 1962], and this approach was used to solve the gradients for any Neural ODE model. [Chen et al., 2018] state the reason it computes the augmented ODE backwards in time is so it can do the job of backpropagation in the discrete-depth networks without storing the intermediate hidden states. In the traditional discrete-depth neural network, we have to store the activations of every layer as we work through the model, in order to compute the backpropogation. In contrast, the ODE solvers starts with an initial state $\mathbf{z}(t_0)$ and computes the trajectory to get to the final state $\mathbf{z}(t_1)$. If we know the final state, we can run the same dynamics backwards in time using the ODE solver. In particular, the original paper by [Chen et al., 2018] supposes that when optimizing a scalar-valued loss function

$$L(\mathbf{z}(t_1)) = L\Big(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\Big) = L(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta))$$

the adjoint $\mathbf{a}(t_i)$ is the quantity of the gradient of the loss with respect to the hidden state $\mathbf{z}(t_i)$ at t_i instant. Suppose t_1 is the final instant, when solving the adjoint states backwards in time, the initial adjoint state is calculated as

$$\mathbf{a}(t_1) = \frac{\partial L}{\partial \mathbf{z}(t_1)}$$

where ∂L is the derivative of the loss and $\partial \mathbf{z}(t_1)$ is the derivative of state $\mathbf{z}$ at time t_1; and the gradient with respect to the hidden state at any time t is

$$\mathbf{a}(t_0) = \mathbf{a}(t_1) + \int_{t_1}^{t_0} \frac{d\mathbf{a}(t)}{dt} = \mathbf{a}(t_1) - \int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}.$$

The gradient of $\mathbf{a}(t)$ with respect to t is the gradient of loss depends on the hidden state $\mathbf{z}(t)$, it is calculated as

$$\frac{\partial \mathbf{a}(t)}{\partial t} = \frac{\partial}{\partial t}\frac{\partial L}{\partial \mathbf{z}(t)} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}.$$

Then we can derive the gradient with respect to θ to optimize the loss L. The gradient is computed by

$$\frac{\partial L}{\partial \theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} dt.$$

Hence there is no need to store the activations, which results in $\mathcal{O}(1)$ memory cost. [Chen et al., 2018] concludes that this approach has advantages of scaling linearly with problem size, having low memory cost, and explicitly controlling numerical error as it computes the integrals. The adjoint method is very important in Differential Equation based neural networks since it provides a method for training the continuous-depth neural network.

3.3 TorchDyn

`TorchDyn` is a python library developed by [Poli et al., 2020] Filling the gap of libraries specifically for continuous-depth models, `TorchDyn` uses neural differential equations and derivative models to make the data as ready-to-use deep learning primitives. This library is designed to present a continuous-depth framework driving transitioning to the modern machine learning framework with `PyTorch`. Core elements for `TorchDyn` includes NeuralDE, DEFunc and Adjoint. The NeuralDE class is primary model class that can interact with `PyTorch`, while the DEFunc class preserve compatibility of NeralDE variants. Also, `TorchDyn` has a complete sensitivity system for neural ODE, which are the standard back-propagation and the adjoint-based gradients, with the capability for gradients to compute integral-loss functions. These primary elements generated by `TorchDyn` are prepared for further coding following by libraries including `PyTorch` and `PyTorch Lightning` which is helpful as we can use their predefined training loop to speed up the training process.

3.4 LSTM Baseline

LSTM was proposed by [Hochreiter and Schmidhuber, 1997] in 1997. According to the lecture notes in week 8, an LSTM network at time step t has an input vector $[h(t-1), x(t)]$, the cell state $c(t)$, and the output vectors $h(t)$. An LSTM has forget gate $g_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t)$; input gate $\tanh(W_{hh}h_{t-1} + W_{hx}x_t) \odot \sigma(W_{ih}h_{t-1} + W_{ix}x_t)$ which controls what new information will be encoded in to the cell state; and output gate $o_t = \sigma(W_{oh}h_{t-1} + W_{ox}x_t)$ which controls what encoded information in the cell state is sent to the network as input. Those three gates give the cell's state $c_t = c_{t-1} \odot f_t + \tanh(W_{hh}h_{t-1} + W_{hx}x_t) \odot \sigma(W_{ih}h_{t-1} + W_{ix}x_t)$ in which the long term dependencies and relations are encoded; cell's output vector $h_t = o_t \odot c_t$. We construct such LSTM model from PyTorch with 300 input size and 300 layer size with 0.1 dropout rate in Python as one of our baseline model. In forward propagates, the sequence outputs and hidden states are obtained from LSTM by PyTorch, then we apply dropout to the sequence outputs. Lastly, we apply linear layers to the sequence output after dropout.

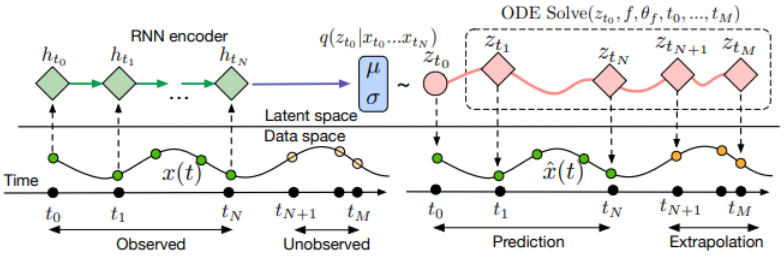
3.5 GPT2 Baseline

According to OpenAI website description, GPT-2, proposed by [Radford et al., 2019], is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages. Firstly, we define a dataset class for train, validate, and test datasets for fine-tuning. The max sequence length is set as 40 since gpt splits up words into smaller tokens. Next, we do fine-tuning of the GPT model using the hugging face out of the box trainer. Specifically, for the training arguments, we set the total number of training epochs since 1 is enough to get very low perplexity and perplexity increases at 2; 16 for batch size per device during training; 64 for batch size for evaluation; the number of warmup steps for learning rate scheduler is 500; and the weight decay is 0.001. We then build a model wrapper for GPT-2 that uses the "past" variable and for language modelling, which outputs the logits for the most likely next word at each position in the sentence and optionally the hidden states. We also do beam search with beam size of 5 to find the most likely sentence.

3.6 ODE-LSTM

In contrast to the ODE-LSTM model by [Lechner and Hasani, 2020] our version of ODE-LSTM simply uses the output of an LSTM as the initial hidden state for an ODE. We first train the LSTM described in section 3.4 on the Penn TreeBank task. Then to perform inference with the ODE-LSTM we run the LSTM on the full input sequence, take the output for the last item and use it as the initial state. Lastly, we apply a linear layer to the hidden states returned by the Neural ODE, before we apply softmax layer to the output sequence to get the prediction results.

Of note is the fact that we run the LSTM on the full input sequence and feed it's last hidden state to the Neural ODE during training. This is different to the original training process in [Chen et al., 2018], shown below



Training process for a RNN-style ODE [Chen et al., 2018]

They propose to evaluate the RNN encoder on only part of the input sequence, passing that hidden state to the Neural ODE which both reconstructs the RNN input sequence and extrapolates the rest of the sequence. However this approach isn't as applicable to language modelling.

In language modelling teacher forcing is used, where the correct words are fed to the sequence model for each input step. If teacher forcing were not used a sequence model would have to generate the entire sentence itself, and predicting a single word wrong would make the predictions for the remaining sentence wrong. This also makes the loss function incorrect, for example if an input sentence were "I went to the supermarket to buy some groceries" and the model predicted "I went to the supermarket to work a shift" then the loss function would penalise the model for predicting "a shift" even though this is a correct generation given the input "work". In summary teacher forcing is an important technique for training language models.

However the original approach of [Chen et al., 2018] is equivalent to training without teacher forcing, as the Neural ODE must extrapolate multiple words of a sentence without any correct input to guide it. Hence we propose to train the Neural ODE for only reconstruction, given the LSTM hidden state for the entire sentence, as an analogue to force teaching. This is because the LSTMs hidden state for the last item contains memory of the entire sequence, and so could be thought of as giving the correct information to the Neural ODE. Though the LSTM may of course misrepresent the sentence we find that this method of training improves the perplexity of all our Neural ODE models.

Finally the actual derivative function used to model the ODE is a simple MLP with one hidden layer and ReLU activation. The layer sizes for are determined by the size of the hidden states, as the size of the output of the derivative function must be the same as the input. In our case that means a 100-dimensional layer for ODE-LSTM and 768-dimensional layer for ODE-GPT.

3.7 ODE-GPT

This model is similar to ODE-LSTM, we define an ODE model that uses GPT-2 [Radford et al., 2019] to get a representation for the sentence which is then passed to the Neural ODE. In feed forward propagation, GPT takes in a single sample and outputs the hidden states for all layers. Since [Tenney et al., 2019] have shown transformers' later layers represent high level meaning, which is what we want to input to the Neural ODE, we then use the output of GPT2's 12th decoder. We feed the 12th decoder into the Neural ODE model to reproduce the output sequence.

3.8 Augmented ODE

Inspired by the augmented differential equation presented in [Dupont et al., 2019] we present our own version. Instead of simply initialising the augmented dimensions as a zero vector as in the original paper we use the hidden state of GPT-2 at the first word. The aim of this is to help with prediction of words near the start of the sentence, it would be helpful for the model to have knowledge of the start of the sequence as well. In general augmented differential equations are useful because they add more dimensions and complexity to the model which should help with the under fitting problem we see in the experiments.

3.9 Implementation Details

We build our Neural Ordinary Differential Equation model as a decoder for language modelling tasks by using the `TorchDyn` [Poli et al., 2020] framework. According to [Poli et al., 2020], we firstly define the `PyTorch` module which takes the role of the Neural ODE vector field f as a neural network. Then we pass the vector field f as an argument into the `NeuralDE` class by `TorchDyn`. We compute backward gradients with the adjoint method discussed above. The adjoint method is implemented by `TorchDyn`. The Neural ODE model uses Dormand-Prince ODE solver by [DOR, 1980] `dopri5` in Python from `torchdiffeq` by [Chen et al., 2018].

In the training steps, we use PyTorch Lightning's training loop. Because of the nature of Neural ODEs we cannot use minibatch training and must instead use stochastic gradient descent. To counter the limitations of stochastic gradient descent, namely the high variance of gradients in each iteration, we use an Adam optimizer [Kingma and Ba, 2017] with a very low learning rate of 0.0001 and higher betas $\beta_1 = 0.95$ and $\beta_2 = 0.999$ to calculate the exponential average over a larger range of gradients.

4 Experiments and Results

4.1 Evaluation

Language modeling is a hard topic to be evaluated in a objective and accurate way at the same time [Chen et al., 1998]. One of the most popular evaluation metric, language perplexity, although it can be calculated efficiently and without access to a speech recognizer, does not correlate well with speech recognition word-error rate. Therefore, we also choose to implement the BLEU (Bilingual Evaluation Understudy), a metric which can also be used to evaluate generated texts and directly connected to the word error rate. With this manner, we believe a more comprehensive insight and conclusion can be drawn from our experiment.

Followings are the details of the two metrics we choose to implement:

- Perplexity
  A popular and standard metric for evaluating the language model. The verb perplex means making someone feel confused. Therefore, the lower perplexity of a language model implies the better performance. According to [Chen et al., 1998], given a language model p_M(next word w|history h) on a test set $T = \{w_1, \ldots, w_t\}$, the perplexity PP(p_M) of the language model calculated is

$$\text{PP}(p_M) = \frac{1}{\left(\prod_{i=1}^{t} p_M(w_i|w_1, \ldots, w_{i-1})\right)^{1/t}}$$

- BLEU (Bilingual Evaluation Understudy)
  A score for comparing the generated sentence of text between one or more reference sentences. Although [Papineni et al., 2002] proposed it for machine translation, BLEU can also be utilised for any kinds of generated texts if enough corpus is provided. BLEU works by counting matching n-grams in the candidate sentence to n-grams in the reference sentences, where 1-gram or uni-gram would be each token and a bi-gram comparison would be each word pair. Note that the comparison is made regardless of word order. Also, according to [Papineni et al., 2002], the BLEU metric ranges from 0 to 1, and 1 means a perfect match, while 0 means a perfect mismatch. However BLEU suffers from the limitation that multiple examples of a candidate sentences are needed to accurately evaluate the performance. As we only have one example, the ground truth sentence, the results from BLEU may not be accurate, which is why we also use the perplexity metric.

4.2 Training

For all four models mentioned in the section 3, we trained and test them on both Penn TreeBank and WikiText2. For GPT, we fine-tuned it corresponding to the dataset it was going to be tested on. To ensure the testing results are fair compared, we calculated the perplexity and BLEU score on the same standard given the same amount of training instances.

Also we use PyTorch Lightning for fetching pre-trained GPT model and its fine tuning, along with fine-tuning the Neural ODE. The predefined training loops of lightning include features such as a learning rate scheduler, and multiprocessing for loading of data, so this helps to speed up the training process.

4.3 Results

The following experiments were conducted using Colab Pro (Intel(R) Xeon(R) CPU @ 2.30GHz, Tesla P100 16GB, 14GB RAM)

Model	Dataset: Penn TreeBank			Dataset - WikiText2		
	Perplexity (train/test)	BLEU (train/test)	# of Params	Perplexity (train/test)	BLEU (train/test)	# of Params
Base-LSTM	212 / 266	**0.295** / **0.228**	3.73 M	171 / 243	0.610 / 0.359	10.7 M
ODE-LSTM	232 / 253	**0.296** / 0.225	3.91 M	224 / 266	**0.612** / 0.357	10.9 M
Base-GPT	**21.57** / **23.95**	0.235 / 0.225	163 M	**15.46** / **16.77**	0.553 / **0.545**	163 M
ODE-GPT	179 / 173	0.177 / 0.179	164 M	303 / 221	0.387 / 0.375	164 M
Augmented-ODE-GPT	169 / 158	0.189 / 0.190	206 M	---	---	---

Table 1. Experiment Results of two evaluation metrics for baseline and NeuralDE models with both train and test performance, trained using teacher forcing.

Firstly, we can obviously spot that the original GPT model performs best in terms of language perplexity for both dataset of Penn TreeBank and WikiText2. Secondly, in terms of BLEU score, things went a little differently. The LSTM prone to do better for Penn TreeBank but overfit a lot in WikiText2 (which will be discussed in 5.1). Concerning only the testing performance on BLEU score, the original GPT model is still the best.

We can see that NeuralDE does not have much help in improving the performance of language modeling in terms of both perplexity and BLEU. On the one hand, the undermining effect is much obvious for GPT compared to LSTM; on the other hand, although we tried the default tuning of hyper-parameters for the ODE solver, the time cost of inference raised significantly (at least twice the training time) according to our experience.

Also, note that the LSTM has an obvious over-fitting for BLEU results on WikiText2, whereas the GPT2 model has done pretraining so it already has a general understanding of language and is less likely to overfit.

4.4 Ablation study

Model	Perplexity (train/test)	BLEU (train/test)
Full Model	**169** / **158**	**0.189** / 0.190
- Zero Intialisation	176 / 165	0.188 / **0.191**
- Teacher forcing analogue	185 / 173	0.182 / 0.186
- No Fine Tuning on GPT	177 / 165	0.185 / 0.185
- Not Augmented	179 / 173	0.177 / 0.179

Table 2. The ablation study for augmented-NeuralDE implemented on GPT for the Penn TreeBank dataset.

We can see that the modifications made, e.g. improving on the zero initialisation of augmented ODEs and training for reconstruction only all have an impact, though it is minor. It is interesting that not fine-tuning GPT-2 only has a minor impact, though in language modelling itself we found fine-tuning to be essential, with the GPT-2 model only achieving 300 perplexity on Penn TreeBank. This indicates that the Neural ODE model doesn't need GPT-2 to be necessarily a good language model itself, only to give embeddings for the sentence that can show its meaning. Though this may also be because the Neural ODE is under fitting and can't take full advantage of GPT-2's embeddings.

5 Discussions

5.1 Why the Performance decreases

The performance of the Neural ODE is overall lacklustre. Though our architecture can actually beat the LSTMs performance in terms of perplexity it falls behind in the BLEU metric. This is because the model has a tendency to predict more common words, if we take the highest probability words for a typical sentence the output tends to look like " the the the N N N N N", lowering the BLEU metric as it fails to match the other words in a sentence. While the correct words also have a higher rating, explaining the lower perplexity, it's clear the model itself still has trouble with language generation. This may simply be due to the nature of an ODE, it may be that a function for generating language is not continuous i.e. it has sharp jumps and changes and hence is not differentiable and cannot be modeled by an ODE. Another issue may be the typical issue that plagues most RNNs, forgetting information from earlier in the sequence. We did experiment with using a GRUCell that has memory as the derivative function, with the hope of increasing the memory of the model, however this suffered from a gradient explosion problem.

5.2 Little affect of using LSTM or GPT

Since GPT is a transformer-based deep learning model with much larger size of parameters compared to LSTM, its capability of fitting and generalising is obviously higher than LSTM. NeuralDE, on the other hand, will only adjust its parameters according to the complexity of its input and labels, which are approximately the same for both LSTM and GPT, given that we use the same dataset. In other words, the resulted ODE has the similar complexity, thus bringing results very close to each other.

5.3 High time cost

Since NeuralDE does not support mini-batch training (also mentioned in the work of [Chen et al., 2018]), we can only train it using the stochastic gradient descent. Therefore, the advantage of GPU for tensor computing is cannot be utilised at all. Given that solver the ODE and dataset are the same for both ODE-LSTM and ODE-GPT, it is understandable that it will both takes very long time.

5.4 Relations between Encoder and Decoder

Since we use an encoder-decoder architecture, where the GPT-2 and LSTM are the encoders and the Neural ODE model is the decoder, the embedding size and complexity of the encoder affect the ODE model performance. We found that increasing encoder complexity does help given the augmented ODE has minor improvements after doubling the embedding size. In the original Neural ODE paper by [Chen et al., 2018], they extrapolate for the unobserved data as the generative output. Training for extrapolation might give improvements, which results in against the conventional wisdom of training language model using teacher forcing. Though it would make sense, because this is not a traditional language model, it must learn to extrapolate without new input. Lastly, we found that there are different derivative functions failed for the encoder. Using a GRU cell as the derivative also failed.

6 Conclusions

The idea of Neural ODEs is newly introduced, and it does not have much applications yet, especially in the field of language modeling. We studied how this idea performs in such undiscovered field by reconstructing a new ODE-LSTM model and proposing a ODE-GPT-2 model for language generating by using the existing Neural ODE framework and other language modeling frameworks. We also compared the performances between our ODE-based language models and the classical state-of-the-art language models. There are limitations in Neural ODEs themselves, so are our models. In the future, we will continue to explore this newly introduced idea of neural ODEs, interesting lines of research include how to implement a memory mechanism as a derivative function and examining whether or not a language modelling function must be continuous. Completing this project has taught us valuable skills, such as understanding of language modelling, how to use pre-trained transformers, designing of new architectures, how to understand research literature and interpret the codebases of new model architectures. Overall this project has greatly expanded our skillset in deep learning.

References

[DOR, 1980] (1980). A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26.

[Alba and Stronov, 2019] Alba, E. D. and Stronov, G. (2019). Neural odes for machine translation.

[Chen et al., 2018] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural ordinary differential equations. *Advances in Neural Information Processing Systems*.

[Chen et al., 1998] Chen, S., Beeferman, D., and Rosenfeld, R. (1998). Evaluation metrics for language models.

[Dupont et al., 2019] Dupont, E., Doucet, A., and Teh, Y. W. (2019). Augmented neural odes. *arXiv preprint arXiv:1904.01681*.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. 27.

[Habiba and Pearlmutter, 2020] Habiba, M. and Pearlmutter, B. A. (2020). Neural ordinary differential equation based recurrent neural network model. *CoRR*, abs/2005.09807.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

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

[Kidger et al., 2020] Kidger, P., Morrill, J., Foster, J., and Lyons, T. (2020). Neural Controlled Differential Equations for Irregular Time Series. *Advances in Neural Information Processing Systems*.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

[Lechner and Hasani, 2020] Lechner, M. and Hasani, R. (2020). Learning long-term dependencies in irregularly-sampled time series. *arXiv preprint arXiv:2006.04418*.

[Marcus et al., 1993] Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

[Merity et al., 2017] Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2017). Pointer sentinel mixture models. *ArXiv*, abs/1609.07843.

[Papineni et al., 2002] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *ACL*.

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543.

[Poli et al., 2020] Poli, M., Massaroli, S., Yamashita, A., Asama, H., and Park, J. (2020). Torchdyn: A neural differential equations library. *arXiv preprint arXiv:2009.09346*.

[Pontryagin et al., 1962] Pontryagin, L. S., Mishchenko, E., Boltyanskii, V., and Gamkrelidze., R. (1962). The mathematical theory of optimal processes.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

[Shen et al., 2020] Shen, X., Cheng, X., and Liang, K. (2020). Deep euler method: solving odes by approximating the local truncation error of the euler method.

[Tenney et al., 2019] Tenney, I., Das, D., and Pavlick, E. (2019). Bert rediscovers the classical nlp pipeline.