AIML 425 | Project
Neural Ordinary Differential Equations

Irshad Ul Ala | Student ID 300397080
Victoria University of Wellington

September 25, 2021

1 Introduction

This paper reviews and discusses the original document titled "Neural Ordinary Differential Equations", a new family of deep neural network models (Chen et al., 2021).
Neural ODEs are neural networks that use the concept of differential equations to parameterize the evolving behaviour of the hidden state of a neural network, in a continuous fashion, versus the convention of discrete, fixed number of hidden layers with linear operators (weight matrices). Please take note that the term **time** will be used very loosely in this paper when describing the parameterization of the hidden states in a neural ordinary differential equation (ODE) system. Time is used effectively as a "buffer" variable to explain the governing parameter, aside from the model parameters (θ). This usage is not in any way meant to suggest that neural ODEs are limited to use for time-variant modelling problems, although, they are conventionally used for such applications.
Neural ODEs are typically used in time-series implementations and not for typical classification tasks. We shall explore the viability of using neural ODEs for classification of a simple dataset of shapes - firstly on singular images (images with one of either 3 shapes : triangle, circle or rectangle), and then compound images (images with either one of 3 shapes, or a unique pairwise combination of the 2).
Additionally, we will substitute the ode solver algorithm that typically uses the explicit Euler's method, with very similar implicit Euler methods, such as implicit Euler's method, and a version of the Crank Nicolson scheme (not the proper version for heat diffusion) for 1st order ODEs. Finally, we shall study how these variants behave with smaller training samples, counteracted with smaller "time" steps taken in the ode solver algorithm, and how viable it may be to reduce training sample, for a finer time step.
Some of the code for the implementation neural ODEs was inspired by the parent paper, as well as other sources (Honchar, 2021) using Pytorch. The code written for this paper was also done so in Pytorch in a similar fashion.
The adjoint method and the ode solver method are initially all directly taken from the parent paper and another available implementation(Honchar, 2021), and then repurposed to behave like a convolutional neural network(CNN) to classify our images. The ode solver method is also going to be mindly altered to experiment with different, but still simple ode solver methods.

2 Theory

2.1 Euler's method & Ordinary Differential Equations

Euler's method is the common solver the differential equation system in neural ODE networks, and is essentially a manipulation of the Taylor series expansion about a point, to the first order.
Differential equations (DEs) are commonly used to describe time or spatially changing physical phenomena, and most do not necessarily have analytical solutions that can be derived by hand. Common examples of partial DEs include the Navier Stokes equation describing forces, Klein Gordon describing wave motion etc.
However, since we are only working with 1 variable, we would call these ordinary differential equations, which can classified to have the form

$$F(x, y, y') = 0 \tag{1}$$

Euler's method involves periodically estimating the curve F based off of the gradient of the function (given by the ODE).

$$y_{i+1} = y_i + f(t_i, y_i) \cdot (t_{i+1} - t_i) \tag{2}$$

where f denotes the first order ODE of interest (hidden layer operation), and y_i and y_{i+1} are the points of the governing, unknown solution we are estimating.

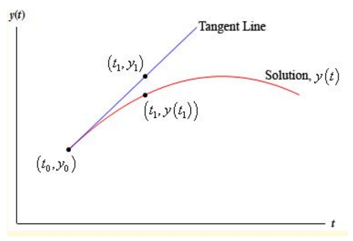

Figure 1: Euler approximation from using tangent

Due to this construction, it is apparent why one needs to begin with a given point (initial condition) on or near the function in order for this approximation to be appropriate.

2.2 Recurrent Neural Networks to Neural ODEs

Considering Euler's method, one can understand the motivation behind the development of neural ODEs.

2.2.1 RNNs use *a priori* information

Recurrent neural networks(RNNs), and residual networks, are distinguished by their use of "a priori" information as they take information from previous inputs to influence the current input and output. In contrast, typical neural networks assume that inputs and outputs with regards to each hidden layer are independent, while the output of recurrent neural networks depend on the prior elements within the sequence(Education, 2020). In fact, one could easily represent the process of an RNN as

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{3}$$

where $f(h_t, \theta_t)$ represents the discrete hidden layer given h_t as an input at time t. It is important to note that an RNN remembers that each and every piece of inputted information depends on time. It is useful in time series prediction because of this time-wary feature that occurs at discrete intervals (layers) - known as long short-term memory. h_t and h_{t+1} are the hidden states at time t and t+1 respectively.

2.2.2 Perspective

The RNN equation could firstly be thought of as an **Euler approximation formulae for time step sizes of 1** (ie $t_{i+1} - t_i = 1$). Let us denote this step by $\Delta t (= 1)$ itself.
Secondly, the layer(s), **instead of being a discrete weighted linear calculation, could be thought to be a continuous hidden state**. An alternative interpretation could be that there are an infinite number of layers between the 2 points in time, and they are all layers generated by one function F.
It is through these 2 conceptualizations that one can claim that a chain of residual blocks in a neural network is basically a solution of the ODE with the Euler's method. Manipulating this formulation, one could easily come up with the following ODE

$$h_{t+\Delta t} = h_t + f(h_t, \theta_t) \cdot \Delta t$$

$$\lim_{\Delta t \to 0} \frac{h_{t+\Delta t} - h_t}{\Delta t} = \frac{dh}{dt} = f(h(t), t, \theta)$$

$$\frac{d\mathbf{h}}{dt} = f(\mathbf{h}(t), t, \theta) \tag{4}$$

It is important to note that the only quantity that changes with time is our hidden state.
With this formulation, we are aiming to specify an ODE, and an initial state of our choosing, such that given an input at time t=0, we train the network to give us the correct derivatives at each point in "time", such that at the end of network, at a time t=T, we obtain the correct destination output.

Thus, instead of having a neural network parameterize a finite set of hidden layers such that we obtain an output from an input, this neural network parameterizes the gradient function describing how the hidden states change continuously such that we obtain the output from the given input- equivalently going through an "infinite" number of hidden states.

2.3 Calculating the loss

A very unique issue that arises from this formulation is that because we have effectively created infinite hidden layers, finding the loss and backpropagating through all of these hidden states would conceivably take a long time. However, this convention would be ill-advised here since ODE-solvers are not perfect, and such a method of backpropagation would likely accumulate more losses over epochs instead of less. Let us instead, set out to find out how the loss varies with the model parameters - $\frac{\partial L}{\partial \theta}$, and use a method called the *adjoint sensitivity method* (Pontryagin et al., 1962).
Additionally, given that $\frac{dh}{dt} = f(h(t), t, \theta)$, it is worth noting that:

$$L(\mathbf{h}(t_1)) = L\left(\mathbf{h}(t_0) + \int_{t_0}^{t_1} f(\mathbf{h}(t), t, \theta) dt\right) \tag{5}$$

2.3.1 How loss, L, varies with hidden state, h(t)

The first step to determining how the loss varies with the parameters is to figure out how it varies with the "infinite" hidden states ($\frac{\partial L}{\partial \mathbf{h}(t)} = \mathbf{a}(t)$), a quantity called the *adjoint*. It is called an adjoint because it is effectively an expression that quantifies how the loss varies due to a **particular point**, given by time dependent h(t). With reference to the adjoint sensitivity method, one can identify that **a**(t) is a curve that satisfies a similar differential equation(Pontryagin et al., 1962)

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t), t, \theta)}{\partial \mathbf{h}} \tag{6}$$

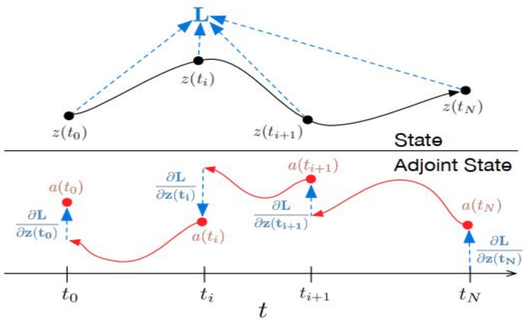
L
$z(t_i)$
$z(t_0)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial z(t_0)}$
$\frac{\partial L}{\partial z(t_i)}$
$a(t_i)$
$a(t_{i+1})$
$\frac{\partial L}{\partial z(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

Figure 2: Normal ODEsolve forward pass versus backwards pass in adjoint method (Chen et al., 2021)

We want to ascertain how the loss varies with respect to the initial value that we have set, $\mathbf{h}(t_0)$, i.e $\frac{\partial L}{\partial \mathbf{h}(t_0)} = \mathbf{a}(t_0)$

because, this model is undergoing training against a set of data that we understand has to obtain a certain endpoint value at t. Hence, the initial value's contribution to the loss must be understood.
Calculating this loss can be done if we apply our ODEsolve algorithm again, but starting from the final point t_1 and with t_0 as our final point, with the given ODE equation in (6). An example of this is shown in Figure 2.
The figure in particular describes a scenario whereby there is more than 1 observation point, not just a simple 1 input, 1 output scenario. In that case, one could simply envision the red adjoint curve being continuous until it reaches t_0 whereby only "$\frac{\partial L}{\partial \mathbf{z}(t_0)}$" would need to be accounted for in order to train the model. Understandably, multiple nodes allow for much more accurate fits.

2.3.2 How loss, L, varies with model parameters,θ

Recall that in this model, the model parameters themselves, inform the model how to go from one hidden state to the next. Once the iteration of the solution is made, $\mathbf{h}(t)$ in the forward pass, and the first iteration of the adjoint state $\mathbf{a}(t)$ is created, the gradient of the loss with respect to the current model parameters is computable with yet another application of the ODEsolver given the equation below

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t), t, \theta)}{\partial \theta} \tag{7}$$

With this set of gradients, the model can effectively be trained per epoch. With the next epoch, a different "infinite" set of hidden states will be defined, from an altered θ, and the loss calculation process repeats itself, coming up with a new set of gradients to train over and so forth, up to a designated point.
Computationally, one would start by establishing an initial value, and final value at times t=0 and t=t, solve for an f initially, and the vector-Jacobian products $\mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t),t,\theta)}{\partial \mathbf{h}}$ and $\mathbf{a}(t)^T \frac{\partial f(\mathbf{h}(t),t,\theta)}{\partial \theta}$ with automatic differentiation concurrently. Then, the quantities $(h)(t), (a)(t)$, and $\frac{dL}{d\theta}$ can be computed together with an ODEsolver based off of the equations (4),(6) and (7).
Interestingly enough, this means that in the course of solving for the loss gradient, one would be seeking to apply the ODEsolve algorithm 3 times, after f is constructed.

2.4 Impact

One of the main impacts is the vast number of advantages that this new form of neural network purports.

2.4.1 Advantages

This method effectively parameterizes the hidden states/layers into a single ODE equation. Instead of storing individual activations and weights in distinct hidden states that constitute a memory cost, the dynamics are simply run backwards from the output at time t, to the starting point at time 0. In other words, there is far lower a memory cost since there is no longer the need for a separate set of independent weight parameters for every layer, while performing at roughly the same level as other networks.

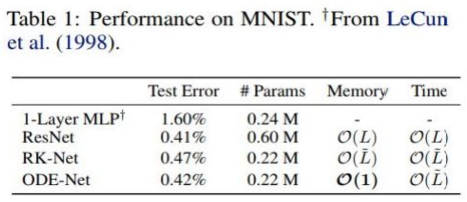
Table 1: Performance on MNIST. †From LeCun et al. (1998).

	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(\tilde{L})$

Figure 3: Memory consumption order comparison between neural networks (Chen et al., 2021)

Secondly, neural ODEs have distinguishable error control. This is simply because the ODEsolver function available has a tolerance level that can be explicitly set by the user. This is especially useful for instance, when switching from training to test data. During training, it would be in one's best interest to typically to train for higher accuracy (lower tolerance), at the cost of large amounts of computation. However, during testing, to minimise the issue of overfitting, one can then choose to switch the tolerance level of the ODEsolver to be higher.

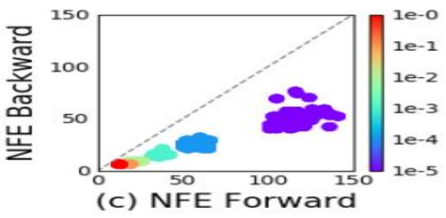

Figure 4: Number of Function evaluations (NFE) consumption comparison between forward and backward pass in ODE-net (Chen et al., 2021)

There is one unexplained phenomena from the parent paper in Figure 4 - the number of computations (function evaluations) in the backward pass of the neural ODE is less(about half) than its own forward pass. This indicates that the adjoint sensitivity method used in neural ODEs also involves **less** computations versus directly backpropagating via the integrator, as it apparently has less computation than the forward pass.
Neural ODE networks also naturally work well with irregularly-sampled time series data, by virtue of their architecture. As evident in Figure 2, one can put in an arbitrary number of groundtruths to compare to for the neural ODE to solve for.

2.4.2 Other application advantages

The parent paper also goes on to list a number of advantages to this method in various applications. One of them including building normalizing flows for density modeling (transformation from one distribution, z_0 by some scale to z_1), by making the typical discrete set of transformative layers, into a continuous transformation, which "simplifies the computation of the change in normalizing constant". Typically, normalizing flows require the differential function f in question

$$\mathbf{z_1} = f(\mathbf{z_0})$$

$$lnp(z_1) = lnp(z_0) - ln\left|det\frac{\partial f}{\partial \mathbf{z_0}}\right| \tag{8}$$

to be one-to-one. However, with a continuous setup, one only requires f to be with a unique solution, as that automatically makes the transformation in the continuous normalizing flow one-to-one. The final log term is also simplified by virtue of instantaneous change of variables($lnp(z_1)$ from $lnp(z_0)$), to be $-tr\frac{df}{d\mathbf{z(t)}}$ - a simpler operation. One of the computational advantages that arise from this is the fact that the reverse transformation of this **continuous normalizing flow** is computationally the same as that of the forward transformation, unlike in the typical normalizing flow setup. As a result, maximum likelihood training is possible to do and one can generate the original distribution of data from a random sample from the resulting distribution z_1, by reversing the continuous normalizing flow, with little computational problem.
Due to the advantage of being able to work with datasets spread out irregularly over time, neural networks can also conversely help to "fill in those gaps" with **generative latent function time-series models**, a type of variational autoencoder. This has also been shown to be extended to recreate typically inhomogeneous Poisson processes such as hospital records, traffic accidents and so forth in the paper. An inhomogeneous Poisson process has a Poisson parameter that is not constant with time, and this in tandem with the latent space that represents the Poisson event of interest can allegedly be solved together with the ode solver in a neural ODE.
Furthermore, the paper also went on to show how a neural ODE could be used to generate much smoother, continuous spirals compared to the typical RNN.

2.4.3 Disadvantages

One of the disadvantages in neural ODEs typically appear to be its lack of compatibility with relatively "simpler" applications, such as fitting to a simple quadratic function.
Another issue with neural ODEs is that they are **overly deterministic**. Every single facet of information given to the network is considered relevant, which explains its aptitude in producing smooth, unique solutions. However, a random, stochastic change to the state that is not in any relevant to the process this network is trying to map would produce a completely different solution. In other words, every single point of data used to train such a network has equivalent leverage, when that genuinely may not be the case. An alternative would be to consider the use of *stochastic differential equations* to account for this sense of randomness to datasets.

2.5 Alternatives to explicit Euler method

As opposed to the explicit euler method (forward method), there is the **implicit euler method (backward method)**, and the **Crank Nicolson scheme (another implicit method)**, that we are going to implement in place of the conventional explicit Euler method in the ode solver. These changes are essentially simple, small adjustments to the point within the time step whereby the gradient is tabulated and used.

2.5.1 Implicit Euler Method | Backwards Euler

One of the main differences between the implicit and explicit euler methods is that

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1}) \cdot (t_{i+1} - t_i) \tag{9}$$

the network is to be evaluated at the unknown point that has yet to be solved for, y_{i+1}, which makes it an implicit method. Such a method is computationally costly (Zeltkevic, 1998).
Despite this, implicit methods are worth consideration because of 2 main reasons. Firstly, implicit techniques are stable, compared to explicit methods(Zeltkevic, 1998). For example, in the case of estimating an initial value problem where $f(t, y) = -ay$ and $y(t = 0) = 1$, having an analytical solution of $y(t) = e^{-at}$, the explicit Euler method the step size **must** be below $\frac{2}{a}$. In contrast, the backwards euler implementation would produce a stable numerical solution of this problem for all conceivable step sizes $(t_{i+1} - t_i > 0)$.
Secondly, the rate of convergence for the implicit Euler method is 2 versus 1 for the explicit Euler method. The global truncation error for both methods however, is notably proportional to h at the final time step, which means that both should similarly decrease or increase in accuracy of results linearly with the time step increase or decrease respectively.

2.5.2 Crank Nicolson implementation

The version of the Crank Nicolson scheme that we will be applying simply involves evaluating the network half-way through the "time-step" taken.

$$y_{i+1} = y_i + f\left(\frac{t_{i+1} + t_i}{2}, y_{i+1/2}\right) \cdot (t_{i+1} - t_i) \tag{10}$$

This is similarly, an implicit method, that we are going to experiment with and evaluate the performance of with our neural ODE network. It is worth noting that this is not the same as taking the average of the forward and backwards euler implementation. It involves using the provided gradient function f (being the model evaluation at that "time" in our case) in the middle of the time step, instead of that at the start or end of the time step.
Since the neural ODEnet method has been shown to be

demonstrably efficient at calculations (thanks to the adjoint method), it seems worth it to attempt to bolster ODEnet's allegedly lackluster performance in image classification, while accounting for additional computation time.

3 Results & Conclusion

The available code for the adjoint method and ode solver are taken directly from the parent paper - specifically the functions named ODEF, LinearODEF, ODEAdjoint, NeuralODE in the code file. From there, all written was meant to repurpose these functions to create a neural ODE to be an image classifier.
Firstly, singular images either containing a triangle, circle or rectangle were tested against a standard neural ODE network with a solver that used the standard, explicit Euler method, with a step-size of 0.05. Then, they were tested against an implicit Euler method solver instead, followed by one using the Crank Nicolson scheme.
All singular object image classification tasks were conducted with an Adam optimizer, of sample size 2700 for training and 6000 for testing. Batch sizes of 120 and 300 were used respectively. Compound image classification was also done with the same sample sizes. However, they were reduced to be trained by 500 images instead of 2700 in the final segment as a "stress" test, as well as a with 1350 images in the Appendix.

3.1 Singular Image classification

3.1.1 Performance Comparison

Crossentropy Accuracy Test Metrics by Epoch			
ODEsolve algorithm	Explicit Euler	Implicit Euler	Crank Nicolson
Epoch 1	43.483%	66.150%	65.933%
Epoch 2	83.883%	95.067%	90.717%
Epoch 3	96.617%	100.000%	94.283%
Epoch 4	99.633%	100.000%	94.067%
Epoch 5	99.633%	100.000%	91.650%

Average training time per epoch was 61 seconds for Explicit Euler network, 71 seconds for Implicit Euler network, and 74 seconds for Crank Nicolson network.

3.2 Compound Image classification

Pytorch specifically does not work with one or multi-hot inputs since they are computationally expensive. Hence, in order to create a mixed dataset of compound (2 objects instead of 1) and singular images, we denote a blank image as '0', triangles as 2, rectangles as 3 and circles as 1. Any combination of 2 non-zero labels was configured to be done by adding an extra 1 to the sum of their labels (e.g. a rectangle and triangle would correspond to 2+3+**1** = 6) ; this makes a unique class label for each possible combination of shapes without much hassle - leading to a total of 7 different classes : blank image, 1 triangle, 1 rectangle, 1 circle, 1 triangle and 1 circle, 1 triangle and 1 rectangle, and 1 circle and 1 rectangle.

3.2.1 Performance Comparison

Crossentropy Accuracy Test Metrics by Epoch			
ODEsolve algorithm	Explicit Euler	Implicit Euler	Crank Nicolson
Epoch 1	61.617%	60.217%	62.333%
Epoch 2	97.550%	77.917%	94.283%
Epoch 3	98.917%	96.033%	98.117%
Epoch 4	99.950%	97.450%	99.800%
Epoch 5	94.567%	99.417%	97.500%

Average training time per epoch was 61 seconds for Explicit Euler network, 66 seconds for both Implicit Euler network and Crank Nicolson network.

3.3 Smaller training sample and step size

Given a much smaller sample to train on, with as proportional a decrease in step taken in their perspective ODEsolve algorithms to compensate, we run the tests again. The following below are results for a step size of 0.01, and training sample of 500- tested over the same 6000 samples.

Crossentropy Accuracy Test Metrics by Epoch			
ODEsolve algorithm	Explicit Euler	Implicit Euler	Crank Nicolson
Epoch 1	33.533%	33.600%	33.400%
Epoch 2	32.850%	33.600%	43.717%
Epoch 3	32.883%	42.067%	33.400%
Epoch 4	62.400%	34.933%	70.533%
Epoch 5	63.717%	43.633%	91.383%

Average training time per epoch was 93 seconds for Explicit Euler network, 81 seconds for both Implicit Euler network and Crank Nicolson network.

3.4 Conclusion

Generally, neural ODEs appear to be able to perform competitively, especially for randomly assorted simple and multiple label image classification. Adding in the implicit Euler method slightly improves the accuracy of image classification (somewhat worse performance for Crank Nicolson scheme). This could lead one to possibly consider favouring the standard implicit method model for a slightly greater computation time.
However, an interesting result arises when the training dataset size is significantly reduced, alongside the step size. Both the explicit and especially the implicit methods significantly deteriorate in performance, despite the step size decrease. In contrast, the Crank Nicolson ODE model appears to still categorise the images quite well (and in less time), indicating that it is perhaps versatile to training with **much** smaller training samples (if sample is not significantly smaller, it will likely be the worst model of the 3 again as shown in section 6.1).

4 Code

Workspace on Colab: `https://colab.research.google.com/drive/12h-2eZYffaOtSwgxN2HUlH-MJ-D9IWJD?usp=sharing`. My contributed code segment starts from the segment titled "Additional Implementation from us". Code for functions ODEF, LinearODEF, ODEAdjoint, NeuralODE earlier on were taken from the parent paper and from Honchar's implementation, with some adjustments.

5 References

Chen, R., Rubanova, Y., Bettencourt, J., Duvenaud, D. (2021). Neural Ordinary Differential Equations. arXiv.org. Retrieved 24 September 2021, from `https://arxiv.org/abs/1806.07366v4`.

Chen, R. (2021). GitHub - rtqichen/torchdiffeq: Differentiable ODE solvers with full GPU support and O(1)-memory backpropagation.. GitHub. Retrieved 24 September 2021, from `https://github.com/rtqichen/torchdiffeq`.

Education, I. (2020). What are Recurrent Neural Networks?. Ibm.com. Retrieved 14 September 2021, from `https://www.ibm.com/cloud/learn/recurrent-neural-networks`.

Honchar, A. (2019). Neural ODEs: breakdown of another deep learning breakthrough. Medium. Retrieved 14 September 2021, from
`https://towardsdatascience.com/neural-odes-breakdown-of-another-deep-learning-breakthrough-3e78c7213795`.

Honchar, A. (2021). Neural-ODE-Experiments/Neural_ODE_Basic.ipynb at master · Rachnog/Neural-ODE-Experiments. GitHub. Retrieved 24 September 2021, from
`https://github.com/Rachnog/Neural-ODE-Experiments/blob/master/Neural_ODE_Basic.ipynb`.

Zeltkevic, M. (1998). Forward and Backward Euler Methods. Web.mit.edu. Retrieved 24 September 2021, from `http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html`.

6 Appendix

6.1 Sample size 1350 | Step size 0.025

Crossentropy Accuracy Test Metrics by Epoch			
ODEsolve algorithm	Explicit Euler	Implicit Euler	Crank Nicolson
Epoch 1	33.700%	51.367%	32.917%
Epoch 2	61.283%	58.667%	32.917%
Epoch 3	66.033%	58.417%	58.100%
Epoch 4	97.900%	96.083%	49.700%
Epoch 5	93.367%	99.883%	90.417%