DVM: A DEEP LEARNING ALGORITHM FOR MINIMIZING FUNCTIONALS

by

Dominic Robert Bair

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Mathematics

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2022

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Dominique Zosso, for not only his guidance and assistance at every stage of this research project, but also his mentorship of me as a student. Without him, I would not have been able to push through my Master's program let alone this thesis. I would also like to thank my committee members, Dr. Jack Dockery and Dr. Scott McCalla. Their input and assistance has helped me to not only ensure this thesis as correct and thorough as possible, but also to better myself as a student of mathematics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The use of data-driven techniques to solve PDEs is a rapidly developing field. Current deep learning methods can find solutions to high-dimensional PDEs with great accuracy and efficiency. However, for certain classes of problems these techniques may be inefficient. We focus on PDEs with a so-called “variational formulation”. Here the solution to the PDE is represented as a minimizer or maximizer to a functional. We propose a family of novel deep learning algorithms to find these minimizers with similar accuracy and greater efficiency than techniques using the PDE formulation. These algorithms can be also be used to minimize functionals which do not have an equivalent PDE formulation. We call these algorithms “Deep Variational Methods” (DVM).

CHAPTER ONE

INSPIRATION

“The curse of dimensionality” is a blanket term used to describe the difficulty of computing the solution to a problem in high dimensions. One area where the curse of dimensionality becomes a particularly difficult problem is in partial differential equations. Partial differential equations (PDEs) have many applications in medical imaging, signal processing, computer vision, remote sensing, electromagnetism, physics, engineering, finance, and more [3, 7, 13]. Traditional finite element methods for computing solutions to PDEs require a mesh. In low dimensional problems, this is not necessarily an issue; however, as the dimension of the problem increases linearly, the size of the mesh increases exponentially. For a PDE with $d \geq 1$ spatial dimensions and 1 time dimension, the size of the mesh must be $\mathcal{O}(n^{d+1})$ [7]. This quickly becomes infeasible due to hardware limitations. With the rise of data–driven techniques to solve high dimensional problems comes new opportunities for addressing the curse of dimensionality.

Deep learning can be an extremely powerful tool when solving problems in high dimensions in certain circumstances and has been used very successfully in areas such as image analysis, biochemistry, and even more recently, mathematics and physics [7, 12, 15]. Deep learning has been applied to a wide variety of problems with seemingly astounding success. It is difficult to find a computational journal without some mention of the topic; but what is deep learning? Let us motivate this with an example.

1.1 Deep Learning

Perhaps we want to create a model that predicts the shoe size of an individual based on their height, weight, sex, age, and/or any number of features we want. Let us say we have D features we consider. We can perform a study with N individual participants and measure each one. So for each individual, i, we have D measurements $\mathbf{x}_i \in \mathbb{R}^D$. Now we concatenate all of these individuals into a data set $\mathbf{X} \in \mathbb{R}^{D\times N}$ where each column of $\mathbf{X}$ is one individual with D measurements, i.e. $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]$. We also measure the shoe size of each of the participants in this study because without that we would have no way of knowing of how well our model works. Let us save the shoe size for each individual i as t_i. Again, we can concatenate these measurements into a data set $\mathbf{t} = [t_1, t_2, \ldots, t_N]^\top$; this is called the target [4, pg 2]. By assumption that the measurements we take are related to shoe size, there is a map $u : \mathbb{R}^D \to \mathbb{R}$ such that $u(\mathbf{x}_i) = t_i$, which we try to infer from the data.

We can use any number of techniques to find an approximation for $u(\mathbf{x})$; call our approximation $y(\mathbf{w}, \mathbf{x})$. Let us consider a one dimensional model, i.e. $D = 1$. For example, we want to relate the height of the individuals in our study to their shoe size; x_i and t_i are the height and shoe size of individual i respectively. We could perform linear regression. That is we assume $u(x)$ depends linearly on x. Therefore $y(\mathbf{w}, x) = w_0 + w_1 x$, where $\mathbf{w}$ is a column vector containing all of the “tunable weights”, $w_0, w_1, \ldots$, in our model. We could also assume $u(x)$ depends non-linearly on x, say a linear combination of M non–linear basis functions $\phi_i(x)$, $i \in 1, 2, \ldots, M$. Our model then would look something like $y(\mathbf{w}, x) = w_0 + w_1\phi_1(x) + w_2\phi_2(x) + \ldots + w_M\phi_M(x)$. We are free to choose what our basis functions, ϕ_i, are; these could be monomials of order i, Fourier functions, or any other basis. However, choosing the basis that will best fit our data can be difficult. Having a fixed basis allows for limited "tunability" of the model. We can, however, approach this problem using a different method.

Any continuous function can be uniformly approximated by a finite number of linear combinations of compositions of a single univariate function; this generalizes to D finite dimensions [14]. These univariate functions can be any non–polynomial function [10]. Consider, for example, a sigmoid function such as hyperbolic tangent. We can uniformly approximate any continuous function as a finite linear combination of hyperbolic tangents. This is an incredibly powerful tool because it removes the need to pick a fixed basis before tuning our model to our data. Using this approach, we can create a model of the form $y(\mathbf{w}, x) = w_{20} + w_{21}\sigma(w_{12}x + w_{10}) + w_{22}\sigma(w_{11}x + w_{13}) + \ldots$ where $\sigma(x)$ is our chosen univariate, non–polynomial function. These weighted $\sigma(x)$'s serve the same purpose as basis functions do in approximating a function; however, these $\sigma(x)$'s are not necessarily linearly independent. What we have just created is a neural network with a single hidden layer, also called a "shallow neural network". The non–polynomial $\sigma(x)$ is called the "activation function". We can see a diagram of a simple shallow neural network in Figure 1.1. In practice, we typically have many more h_i nodes in these models. The specific number of these nodes needed to uniformly approximate a given $u(x)$ up to a given error is, in general, not known but is assumed to be very large [14]. In practice, a sufficient number is often found by trial–and–error. The number of nodes in a layer is called the layer's "width".

A shallow neural network is the simplest form of "deep learning". It takes in at least one input, one node for each dimension of our input data $\mathbf{x}_i$. In our shoe size example, this is all of the measurements $\mathbf{x}_i$ of an individual i. The network output is the model prediction of t_i, in our example, shoe size of individual i.

We have encountered our first neural network; however, networks with shallow architecture are often inefficient when solving problems. Continuing to add more

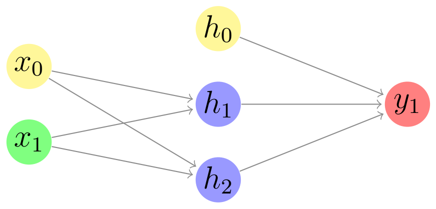

Figure 1.1: Shallow Neural Network: The network takes an input x, applies a bias term, 1, at the node x_0, applies weights $\mathbf{w}$ represented by the arrows in the diagram, sums these weighted terms and applies a non–polynomial activation function $\sigma(x)$ at the nodes h_1 and h_2, then adds another bias term, 1, at the node h_0, applies more weights to each of these terms and finally sums these to get $y(x)$ which is our approximation of u. Our model is thus $y(\mathbf{w}, x) = w_{20}+w_{21}\sigma(w_{12}x+w_{10})+w_{22}\sigma(w_{13}x+w_{11})$

.

nodes to the hidden layer can have diminishing returns on the accuracy of the model; recall the number of nodes to uniformly approximate a given function up to given error is assumed to be very large.

The next step is to add more "depth" to the model; that is, more hidden layers as seen in Figure 1.2. This is called a deep neural network. The distinction between deep and shallow architecture is distinguished by the number of hidden layers in the model[1].

There are innumerable architectures that can be chosen for a model. The models we use for the remainder of this dissertation are all feed–forward networks with fully–connected layers such as seen in Figure 1.1 and 1.2. Deep neural networks use linear combinations and compositions of non–polynomial univariate functions as

[1]There is no standard on the number of hidden layers distinguishing deep and shallow architectures; however, a network with more layers is "deeper" and a network with fewer layers is "shallower".

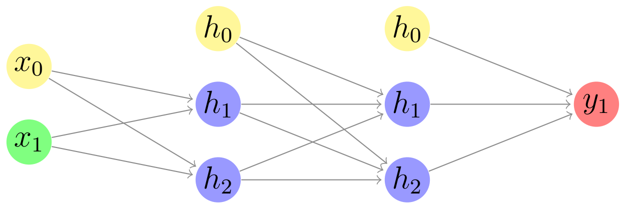

Figure 1.2: Deep Neural Network: This network is one layer deeper than the network seen Figure 1.1. The network output for this model is $y(\mathbf{w}, x) = w_{30} + w_{31}\sigma(w_{20} + w_{22}\sigma(w_{10}+w_{12}x)+w_{24}\sigma(w_{11}+w_{13}x))+w_{32}\sigma(w_{21}+w_{23}\sigma(w_{10}+w_{12}x)+w_{25}\sigma(w_{11}+w_{13}x))$. Notice this network has compositions of our non–polynomial activation function $\sigma(x)$.

approximations, rather than just linear combinations as was the case in shallow neural networks. Relating back to basis functions, each layer forms something akin to a basis for the next layer. In essence, our model is a linear combination of compositions of something similar to basis functions. We reiterate that these functions are not a true basis as they are not necessarily linearly independent from one another. The increased non–linearity of our model allows for more "tunability" of the weights in our model than in shallow architectures. The number of layers in a network is called its "depth". Increasing the depth of a network can yield more accurate results with less nodes; however, the number of model weights increases faster than when increasing the width of the network.

Increasing the model depth also makes it difficult to understand what the model is. This can lead to complications when the model performs poorly because it is difficult to understand where the model "goes wrong". This leads to deep neural networks being referred to as "black boxes" as they are still not well understood [21].

Regardless of which model we pick, neural network or otherwise, we would like it to be accurate. Going back to the shoe size example, if an individual had measurements $\mathbf{x}_i$, we want the model to give an output as close to t_i as possible. We therefore need to optimize or “train” the model to best fit our data.

1.1.1 Loss Functions

Consider any model $y(\mathbf{w}, \mathbf{x})$ with $\mathbf{w} \in \mathbb{R}^k$, meaning there are k weights in the model, $\mathbf{x} \in \mathbb{R}^D$, and $y(\mathbf{w}, \mathbf{x}) \in \mathbb{R}$. One way we could optimize our model is by finding the parameters that give us the maximum likelihood estimate (MLE). Real data is often messy so we cannot just assume $y(\mathbf{w}, \mathbf{x}) = t_i$, so let us assume that our target variable, t_i has some added Gaussian noise. Thus $t_i = y(\mathbf{w}, \mathbf{x}_i) + \epsilon$, where ϵ is a zero mean Gaussian random variable with variance σ^2 [4, pg 140]. Now we want to find the model weights $\mathbf{w}$ that maximize the likelihood of observing the target $\mathbf{t}$. So we obtain the following likelihood function:

$$p(t|\mathbf{w}, \mathbf{x}, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(t_i|y(\mathbf{w}, \mathbf{x}_i), \sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{(2\pi\sigma^2)}} \exp\left(-\frac{1}{2\sigma^2}(y(\mathbf{w}, \mathbf{x}_i) - t_i)^2\right).$$

We would like to pick the weights, $\mathbf{w}$, that maximize this likelihood function, i.e:

$$\mathbf{w}_{MLE} = \underset{\mathbf{w} \in \mathbb{R}^k}{\arg\max} \prod_{i=1}^{N} \frac{1}{\sqrt{(2\pi\sigma^2)}} \exp\left(-\frac{1}{2\sigma^2}(y(\mathbf{w}, \mathbf{x}_i) - t_i)^2\right).$$

We can rewrite this as a minimization problem by taking the negative logarithm of the likelihood function to obtain

$$\mathbf{w}_{MLE} = \underset{\mathbf{w} \in \mathbb{R}^k}{\arg\min} \sum_{i=1}^{N} \ln\left(\frac{1}{\sqrt{(2\pi\sigma^2)}}\right) - \frac{1}{2\sigma^2}(y(\mathbf{w}, \mathbf{x}_i) - t_i)^2.$$

To find the minimizers, we can omit the constant terms, since they do not affect where minimizers occur. So we are just left with

$$\mathbf{w}_{MLE} = \underset{\mathbf{w} \in \mathbb{R}^k}{\arg\min} \sum_{i=1}^{N} (y(\mathbf{w}, \mathbf{x}_i) - t_i)^2.$$

This tells us that our model minimizes the squared error of our data when using $\mathbf{w}_{MLE}$. Thus, our model, $y(\mathbf{w}_{MLE}, \mathbf{x})$, has been trained to fit the data as best as it can. Note this does not mean our model is the best fit for our data. Choosing different model "hyperparameters", such as using a linear regression model, using a neural network, the choice of network depth and width, and the choice of activation functions can lead to a better model. The process of constructing and optimizing such models is called machine learning[2]. The function we have arrived at is called the error or "loss function". These functions are not unique. It is common practice to normalize this function by the number of points. It is also common that we have some prior information that we can include in the model. For example, we may know the model cannot be negative, so we would add a term to penalize a negative output. This changes our model from being a maximum likelihood estimate, to a "maximum posterior estimate"[4, pg 30]. Since we are only interested in optimizing the model, we typically start at defining a loss function rather than a likelihood function since they have the same minimizers and the loss function is simpler in that it is a sum rather than a product. However, minimizing a loss function is often easier said than done. If our model was a neural network, there could be billions of weights to optimize, so finding the minimizers explicitly is intractable.

1.1.2 Optimizing the Model

We would like to find the weights that minimize our loss function. With potentially billions of weights, it can be difficult to find critical points, and more difficult yet to find the global minimizer explicitly. So we must devise a numerical method to find the best approximation of the global minimizer as we can.

[2]This specific example is called supervised learning because we know what the model outputs should be and we penalize it for giving a different output. In unsupervised learning, we do not necessarily know what the output should be. In a clustering problem, for example, we may not even know how many clusters the model should find [4].

An immediate choice is a gradient descent algorithm. While these algorithms vary in implementation, they follow the same core rules: evaluate the function which we wish to optimize at a chosen point, in this case our loss function; calculate the direction of greatest descent at this point[3], in this case with respect to all of our weights; choose a step size; make a step in the direction of greatest descent and allow that to be our new point; repeat the process until the function is "sufficiently" minimized[4].

We must first discuss what it means for a function to have a single global minimum. We introduce the concept of convexity. A set $\mathbf{X}$ is said to be "convex" if $\forall\, x, y \in \mathbf{X}$, and $\forall\, \alpha \in [0,1]$, we have $\alpha x + (1-\alpha)y \in \mathbf{X}$. In other words, if we pick any two points in $\mathbf{X}$ and draw a straight line segment between them, every point on the line segment is also in $\mathbf{X}$ [5, pg 48]. Now consider a function $f : \mathbb{R}^D \to \mathbb{R}$. The "graph" of f is the set of all points such that $\{(\mathbf{x}, f(\mathbf{x})) : \mathbf{x} \in \mathbb{R}^D\}$. We can think of this as simply the plot of $f(\mathbf{x})$ versus $\mathbf{x}$. Now consider the region "above" the graph of f: The "epigraph" of f is the set of all points such that $\{(\mathbf{x}, \beta) : \mathbf{x} \in \mathbb{R}^D,\ \beta \geq f(\mathbf{x})\}$. Now finally we say the function f is "convex" if the epigraph of f is convex [5, pg 512].

If f is convex, it has at most one global minimum. If f is convex and has multiple minimizers, every $f(\mathbf{x})$ along some straight line connecting the minimizers must be equal to the global minimum. Having a function with this property ensures that our gradient descent algorithms will converge to the global minimum.

Let us now look at the most basic form of gradient descent on a convex function. Consider a function $f : \mathbb{R}^D \to \mathbb{R}$ and some direction $\mathbf{d} \in \mathbb{R}^D$ such that $\|\mathbf{d}\| = 1$.

[3]We will use automatic differentiation to find the gradient. This algorithm is discussed in Section 1.1.3.

[4]"Sufficiently" minimized will depend on the specific circumstance. Often we wish to avoid over–fitting our model so that it can be applied more robustly to data outside of our training set.

The inner product $\nabla f(\mathbf{x})^{\top}\mathbf{d}$ gives us the rate of increase of f at the point $\mathbf{x}$ in the direction $\mathbf{d}$. By the Cauchy–Schwartz inequality we have $\nabla f(\mathbf{x})^{\top}\mathbf{d} \leq \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{d}\| = \|\nabla f(\mathbf{x})\|$. Now let $\mathbf{d} = \dfrac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$. Then $\nabla f(\mathbf{x})^{\top}\dfrac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|} = \|\nabla f(\mathbf{x})\|$.

By the Cauchy–Schwartz inequality, this is the largest the inner product can be. Thus, the direction $\nabla f(\mathbf{x})$ is the direction of greatest rate of increase of f at $\mathbf{x}$. Therefore, the direction $-\nabla f(\mathbf{x})$ is the direction of greatest rate of decrease of f at $\mathbf{x}$.

Now consider some starting point $\mathbf{x}^0$ and some other point $\mathbf{x}^0 - \alpha\nabla f(\mathbf{x})$, where $\alpha > 0$. Note that this second point is in the direction of greatest descent from our starting point. By Taylor's Theorem we have $f(\mathbf{x}^0 - \alpha\nabla f(\mathbf{x}^{\mathbf{0}})) = f(\mathbf{x}^0) - \alpha\|\nabla f(\mathbf{x})\|^2 + o(\alpha)$. Thus for sufficiently small α, we have $f(\mathbf{x}^0 - \alpha\nabla f(\mathbf{x})) < f(\mathbf{x})$.

This means our second point $\mathbf{x}^0 - \alpha\nabla f(\mathbf{x})$ is closer to a minimum than our starting point $\mathbf{x}^0$, assuming the function is convex. We can use this to form an algorithm to find an $\mathbf{x}$ that is as close to the true minimizer as we can get. Consider the point $\mathbf{x}^k$. To move to the next point $\mathbf{x}^{k+1}$, we move by an amount $-\alpha_k\nabla f(\mathbf{x}^k)$, where $\alpha_k > 0$. This leads to the following procedure:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k\nabla f(\mathbf{x}^k).$$

The next thing we need to consider is our choice of α_k. We want the maximum amount of decrease at each step, so we choose the following α_k:

$$\alpha_k = \arg\min_{\alpha>0} f(\mathbf{x}^k - \alpha\nabla f(\mathbf{x}^k)).$$

This problem is a standard one–dimensional optimization problem and can be solved using one of many standard techniques called "line search" algorithms [5, pg 131–133].

We now have a way of numerically finding the global minimizer to a potentially high–dimensional, convex function. However, there is a glaring issue with the standard gradient descent algorithm. Gradient descent will only converge to a local minimimum. What if our function has multiple local minima? Our loss function is dependent on possibly billions of weights, so it is difficult to guarantee there is a single local minimum. So we need more robust techniques.

Recall that our loss function is dependent on points from a data set. Depending on the size of our data set, it may be unreasonable to evaluate the loss with the entire data set. To get around this issue we sample a subset of the data points and evaluate the loss with just this "mini–batch". Consider our data set $\mathbf{X}$ and target set $\mathbf{t}$ from our shoe size example. There are N individuals in this data set. Let us take a subset of $M < N$ of these individuals. Typically, we randomly select this subset, but that is not strictly necessary. Call this mini–batch $\mathbf{X}_{M,0}$ and target mini batch $\mathbf{t}_{M,0}$. We can evaluate our loss function using these data points to get:

$$E = \sum_{i=1}^{M} |y(\mathbf{x}_i) - t_i|^2.$$

Now we can calculate the gradient of our loss function with respect to our model weights. Then we update our model weights $\mathbf{w}$ according to our gradient descent algorithm. Now we take a new subset of our data set and target data set, call them $\mathbf{X}_{M,1}$ and $\mathbf{t}_{M,1}$ respectively and repeat the gradient descent algorithm until our loss is sufficiently minimized. This is called stochastic gradient descent (SGD)[5]. SGD has the advantage that if our initial mini–batch $\mathbf{X}_{M,0}$ was chosen "poorly", that is, chosen such that gradient descent converges to a local minimum instead of the global minimum, a new mini–batch is selected $\mathbf{X}_{M,1}$ immediately for the next step which

[5]Specifically this is mini–batch stochastic gradient descent.

will hopefully help the algorithm converge to the true global minimum. It should be noted that each new batch we select begins what is called a new “epoch”. Often we select a batch, then select mini–batches from the batch and perform gradient descent on our loss function evaluated at all points of the mini–batch. Each time we select a new mini–batch begins what is called a new “iteration”. Once we iterate over all mini-batches, we begin a new epoch by selecting a new batch [18].

There is another trick we can use to help keep our algorithm from getting stuck away from the global minimum. If we add momentum to our algorithm, it can “climb” out of local minima, avoid the issue of vanishing gradient, and give the algorithm a greater chance of reaching the global minimum. Momentum in this sense is simply adding a push in the direction from our previous step of gradient descent to our next step in the algorithm. There are a large number of algorithms that have been developed that do this such as SGDM, ADAGRAD, RMSP, ADAM, NADAM, and FTRL [17]. These algorithms vary substantially from one another; however, they all provide momentum to the gradient descent algorithm.

Regardless of the gradient descent algorithm we choose, we need some way of calculating the gradient of the loss function. We now introduce some software we use to accomplish this, as well as other deep learning tasks.

1.1.3 TensorFlow

TensorFlow is a machine learning library with a Python interface called Keras. The package includes a substantial number of linear algebra, data manipulation, and deep learning tools and was developed to allow users the ability to use pre–made functions for simple projects while also allowing for flexibility in advanced projects. TensorFlow is a very popular choice for deep learning projects because of its simple interface and adaptability to different applications [1]. We use Tensorflow with the

Logistic Function	$\frac{1}{1+e^{-x}}$
Hyperbolic Tangent	$\tanh(x)$
Inverse Tangent	$\arctan(x)$
ReLU	$\max\{0,x\}$
ELU	$\left\{\begin{array}{ll} x & \text{if } x>0 \\ \alpha(e^x-1) & \text{if } x\leq 0 \end{array}\right\}$

Table 1.1: A table of commonly used activation functions.

Keras interface for all worked examples in this dissertation.

Every calculation done on a computer, no matter how complicated, is a composition of simple, but possibly many functions. We can store these functions in a list and once the entire calculation is complete we can apply the chain rule to the list of functions to find the exact gradient of any calculation made of compositions of supported functions. This is called automatic differentiation and there are many different software packages, including Tensorflow, that have their own automatic differentiation functions; each of them have a list of supported functions that can be differentiated explicitly.

1.1.4 Activation Functions

Before we jump into an example, we need to further discuss the non–polynomial activation functions at the hidden nodes in a neural–network. Each hidden node applies a non–polynomial transformation to the sum of the inputs of the node. Recall that any function can be approximated with a neural network using any non–polynomial activation [10]. However, the best choice of that non–polynomial activation depends strongly on the situation. We list a few commonly used activation functions in Table 1.1.4.

If we know our model, $y(\mathbf{w}, \mathbf{x})$, needs to be smooth in $\mathbf{x}$, it would be wise to select an activation function that is smooth, such as a sigmoid, or hyperbolic tangent. Recall a one–layer neural network approximates a function as a linear combination of the activation functions. If we choose an activation function such as ReLU, our model would be effectively a linear combination of piecewise linear functions [2]. This would be a poor choice to approximate a smooth function.

1.1.5 Learning a Model of One Variable

We finally have all of the information we need to create our first neural network. First, let us create a data set. Consider a uniform random sample of 2500 points from the interval $[0, 4\pi]$. Let these points be our $\mathbf{X}$. Now let our target $\mathbf{t}$ be defined by $t_i = \sin(x_i) + \epsilon_i$, where each ϵ_i is a random variable selected from a zero–mean, normal distribution with variance 0.1; this will serve as our noise. In a real situation, we obviously would not have a formula for t_i; we would simply have a list of data, but we would expect the target data to have some noise associated with it. So now we construct and train a neural network using Keras.

In the code, we have called our x_i's, "x_train" and our t_i's, "t_train". When we run the "model.fit()" line, the network begins training the model and displays the loss after each epoch. See Listing 1.1.5. Note the loss begins as 0.5429 and ends at 0.0137. See Listing 1.1.5. We specifically choose the tanh activation function for our nodes because we have prior knowledge of our data and expect a smooth model. Note this is a shallow neural network since it has only a single hidden layer. We can see the plot of our neural network and sampled test data in Figure 1.3.

It is easy to see the network did not perform exceptionally well. So the next step is to add more hidden layers, or train the network for longer; we stay with 200 epochs since a simple model like this should converge quickly. Using four hidden layers, Listing 1.1: High–Level Deep Learning Using Tensorflow

```
model = Sequential()
# Creates a feed—forward neural network
model.add(Dense(100, input_dim=1, activation='tanh'))
# Creates one hidden layer with 100 nodes and uses tanh
# activation
model.add(Dense(1)) # Creates output layer
model.compile(loss='mean_squared_error', optimizer='adam')
# Selects loss function and gradient descent algorithm
model.fit(x_train, t_train, epochs=200, batch_size=50)
# Trains the model by selecting batches of 50 points and
# 200 training epochs
```

Listing 1.2: Training output from code in Listing 1.1.5

```
Epoch 1/200
50/50 [==============================] - 0s 1ms/step - loss: 0.5429
Epoch 2/200
50/50 [==============================] - 0s 1ms/step - loss: 0.4405
...
Epoch 200/200
50/50 [==============================] - 0s 1ms/step - loss: 0.1037
```

each with the 100 nodes and using tanh activation, we see significant improvement in the model as seen in Figure 1.4. We see the loss is significantly lower with the deeper neural network with a final loss of 2.1246×10^{-4}; this is an improvement of three orders of magnitude. Note that in this instance, the loss is explicitly the MSE between our last set of training points, and our model.

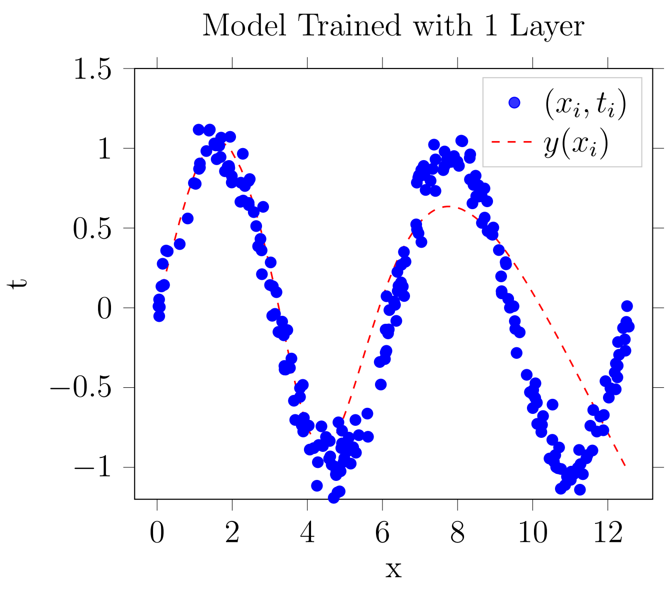

Figure 1.3: The model trained with 1 hidden layer along with test data. Note the very visible error.

1.1.6 Learning a Model of Two Variables

We can use a similar neural network to learn a function of two dimensions as well. We only need to change the "input_dim" argument to 2. Consider a set of data points distributed on an annulus with inner radius 2 and outer radius 4. For this example, our x_i's will have the form (r, θ). Now define our target t_i's by the following function:

$$t(r,\theta) = \left(\frac{4}{1023}r^5 - \frac{4096}{1023}r^{-5}\right)\sin(5\theta) + \epsilon$$

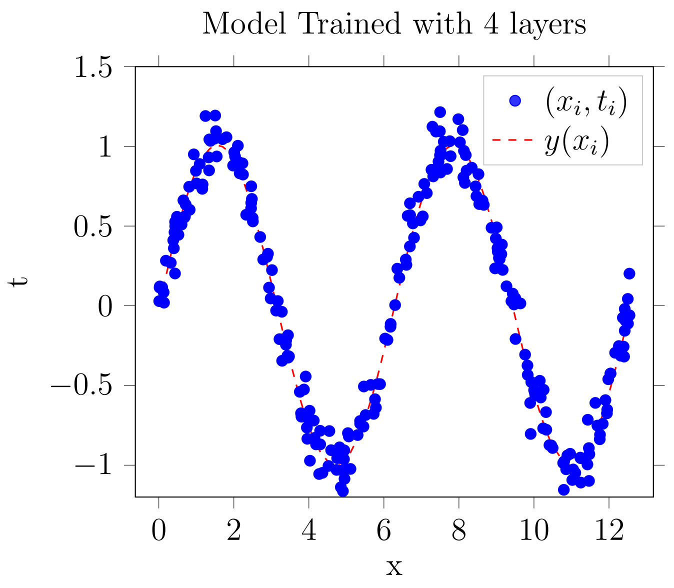

Figure 1.4: The model trained with 4 hidden layers along with test data.

Again, ϵ is a random variable distributed by a zero–mean Gaussian with variance 0.1. To generate uniform data on an annulus we need to be a bit more careful about how we pick our x_i's. If we simply uniformly picked a radius from the interval $[2, 4]$ and an angle from the interval $[0, 2\pi)$ we would get clustering near the smaller radius. To correct this, we choose r such that

$$r = \sqrt{z(4^2 - 2^2) + 4^2},$$

where z is a uniform random variable sampled from the interval $[0, 1]$. So now we can generate our data pairs $(\mathbf{x}_i, t_i)$, and train the model. We see the trained model in Figure 1.5. After 200 epochs, the loss is 0.0164. Notice that the loss is higher than the one–dimensional model with the same network structure and same noise in our training t_i's. The loss is again the MSE between our last set of training points and the model.

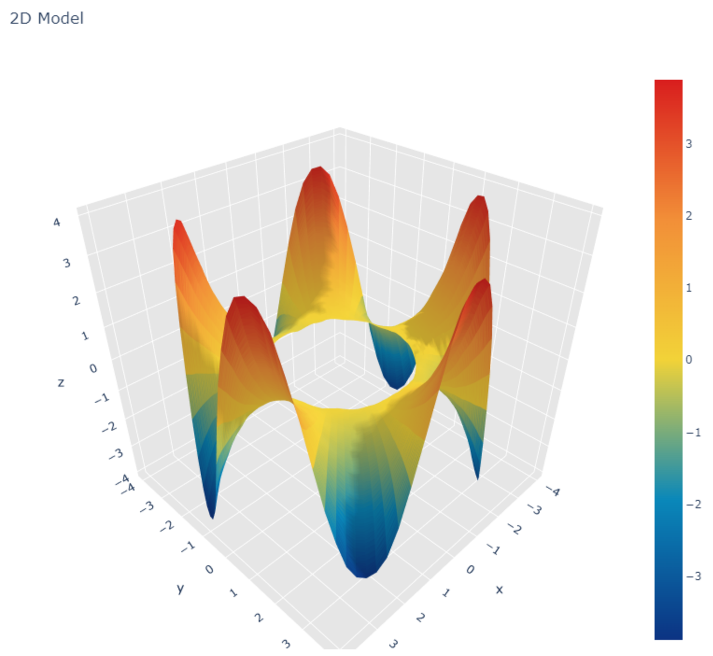

Figure 1.5: Model trained on two–dimensional data. The test points are omitted for easier visualization of the learned model.

1.2 Partial Differential Equations

The models we have looked at thus far have been trained using supervised learning. We knew the model should give an output close to the target observations we have already made. What about situations where we do not have a target data set? The examples we looked at had data that was defined by some function with added noise, the network essentially learned the underlying function without the noise. So can we use this to learn other functions we don't explicitly know? For example, consider a PDE with boundary conditions, initial conditions, potentially other conditions, and a differential operator. We know the function that solves this differential equation must satisfy all of these conditions, but often we cannot explicitly solve for this function. We must therefore rely on numerical techniques such as finite differences which require a mesh. However, as the dimension of this equation increases, the ability to efficiently generate a mesh becomes an intractable problem [9], so can we use a data–driven technique to solve these equations? This is where "Deep Galerkin Methods" (DGM) come in [7]. As in our two previous examples, the network is trained on randomly selected points from the domain. This completely removes the need for a mesh [7]; thus, allowing for a solution to be found in these high–dimensional cases, while also allowing us to consider moving–boundary problems [9].

CHAPTER TWO

DEEP GALERKIN METHODS

We discuss the general ideas of DGM as introduced in [7] by example. Consider the following PDE:

$$\begin{cases} \mathbf{D}u(\mathbf{x},t) = f(\mathbf{x},t), & (\mathbf{x},t) \in \Omega \times [0,\infty) \\ u(\mathbf{x},t) = g(\mathbf{x}), & (\mathbf{x},t) \in \partial\Omega \times [0,\infty) \\ u(\mathbf{x},0) = h(\mathbf{x}), & (\mathbf{x},t) \in \Omega \times \{0\} \\ \vdots \end{cases} \tag{2.1}$$

We have a differential operator $\mathbf{D}$ which is defined on some open domain Ω, boundary condition $g(\mathbf{x})$ on $\partial\Omega$, initial condition $h(\mathbf{x})$, and potentially other conditions. How do we train a neural network to solve this problem?

First we must verify the PDE is well posed. That is: there exists at least one solution; there exists at most one solution; the solution depends continuously on the data [13, pg 6]. We also must verify the PDE has a continuous solution. Recall that neural networks can uniformly approximate continuous functions [14]; this does not necessarily hold for discontinuous functions. PDE's, in general, are not well posed as they may not have a solution; they may have multiple solutions; their solution may not depend continuously on the data. For the purposes of this dissertation, we only consider well posed problems with continuous solutions.

Once we verify that (2.1) is well posed, and has a continuous solution, we proceed to create a model. Denote $y(\mathbf{w},\mathbf{x})$ as our model of $u(\mathbf{x})$. Let us begin by taking a random sample of L points from $\partial\Omega \times [0,\infty)$. We know that these points must satisfy $y(\mathbf{w},\mathbf{x}_i,t_i) = g(\mathbf{x}_i)$ so we penalize the network for not satisfying this condition. Next, we take a random sample of M points from $\Omega \times \{0\}$. These points must satisfy $y(\mathbf{w}, \mathbf{x}_i, 0) = h(\mathbf{x}_i)$, so we penalize the network for failing to meet this condition. We continue this process for each of our boundary and initial conditions. Now we need to consider the differential operator. We take a random sample of N points from $\Omega \times (0, \infty)$; in practice we of course truncate time to a finite number. These points must satisfy $\mathbf{D}y(\mathbf{w}, \mathbf{x}_i, t_i) = f(\mathbf{x}_i, t_i)$. So we again penalize the network for not satisfying this condition. We can now formulate a loss function to train our model $y(\mathbf{w}, \mathbf{x}, t)$ on:

$$\begin{aligned} E = \frac{1}{L}\sum_{i=1}^{L} |y(\mathbf{w}, \mathbf{x}_i, t_i) - g(\mathbf{x}_i)|^2 + \frac{1}{M}\sum_{j=1}^{M} |y(\mathbf{w}, \mathbf{x}_j, 0) - h(\mathbf{x}_j)|^2 + \cdots \\ + \frac{1}{N}\sum_{k=1}^{N} |\mathbf{D}y(\mathbf{w}, \mathbf{x}_k, t_k) - f(\mathbf{x}_k, t_k)|^2. \end{aligned} \tag{2.2}$$

Recall that we wish to find the model weights that minimize this function. When the model is well trained, this loss E will be near 0. The choice of L, M, and N is somewhat arbitrary and we must, unfortunately, rely on trial–and–error to determine adequate values. Note that each of the terms in (2.2) are discrete approximations for continuous L_2 errors on the boundary and interior [20].

We have already discussed that optimizing the model requires calculating the gradient of the model with respect to the model weights $\mathbf{w}$, but how do we calculate the differential operator at the point $(\mathbf{x}_k, t_k)$? Recall that in automatic differentiation, we store every operation performed. This means we can take the derivative with respect to any variable that passed to the stored functions. Since the model $y(\mathbf{w}, \mathbf{x}_k, t_k)$ is a function of our model weights $\mathbf{w}$ as well as the point $(\mathbf{x}_k, t_k)$, we can use automatic differentiation to calculate the derivatives of the model with respect to $\mathbf{x}_k$ and t_k. For higher order derivatives, we can simply call automatic differentiation multiple times, since automatic differentiation is itself a function which we can take the gradient of. This can be a relatively expensive process since a neural network is parameterized by potentially billions of compositions of functions. Despite this, these methods have shown considerable accuracy in certain high–dimensional problems; [7] demonstrates a neural network approximating the solution to the Black–Scholes equation in 200 dimensions with a 0.22% relative error. We now delve into some examples.

2.1 The Laplace Equation

2.1.1 In One Dimension

Consider the following PDE:

$$\begin{cases} \Delta u(x) = 0, \quad x \in (0,1) \\ u(0) = 0 \\ u(1) = 1 \end{cases} \tag{2.3}$$

This Laplace equation has the unique solution $u(x) = x$ [13, pg 117]; thus, it is well posed and makes it a perfect starting example for studying the effectiveness of DGM. We first need to alter (2.2) for this specific example. We only have boundary conditions and a differential operator to satisfy; however, our boundary is disconnected, so we need three terms in the loss function.

$$E = \lambda(|y(\mathbf{w},0) - 0|^2 + |y(\mathbf{w},1) - 1|^2) + \frac{1}{N}\sum_{i=1}^{N} |\Delta y(\mathbf{w}, x_i) - 0|^2 \tag{2.4}$$

Note that our boundary consists of only two points, so random sampling from each condition makes little sense. We instead apply weight λ to the boundary conditions to give them a similar weight as the term penalizing the model for not satisfying the differential operator. We find this λ term is unnecessary; however initial experiments showed it may be beneficial. We also normalize by the number of points we select to find the mean square error (MSE) of the last term in (2.4).

We now need to pick a network architecture. We start with a feed–forward network with five hidden fully–connected layers and 100 nodes in each layer. We know the solution is smooth since it is harmonic, so we want to pick a smooth loss function. We choose hyperbolic tangent for the activation in each of our hidden layers. For training, we choose 20 epochs each with 100 iterations, and we randomly select 100 points from the domain $(0, 1)$ in each iteration to train the network on. We also choose $\lambda = \sqrt{10}$. We see the resulting model in Figure (2.1). After training the model, we randomly sample another 100 points from the domain and find the mean absolute error (MAE) of the model evaluated at these points to be 5.80302×10^{-3}. If we change the hyper–parameters of our model we can get significantly different results. See 2.1.1 for a full table of hyper–parameters we modify and the resulting final MAE. In all combinations, we use 100 iterations in each epoch with 100 randomly sampled points from the interval $(0, 1)$.

We see that despite ReLU being the solution to our problem on the interval $[0, 1]$, the tanh activation provides a solution one to two orders of magnitude better than ReLU with all other hyper–parameters being the same. There is variation in our MAE. Selecting another batch of 100 points will yield different results because of the stochastic nature of our method. Notice that with a deeper network, our final MAE was actually worse in all examples than the shallower network. This is likely due to the simplicity of the solution. Trying to approximate a straight line with combinations and compositions of non–linear functions is not the best way of approaching this problem, but it gives us insight into hyper–parameters we should

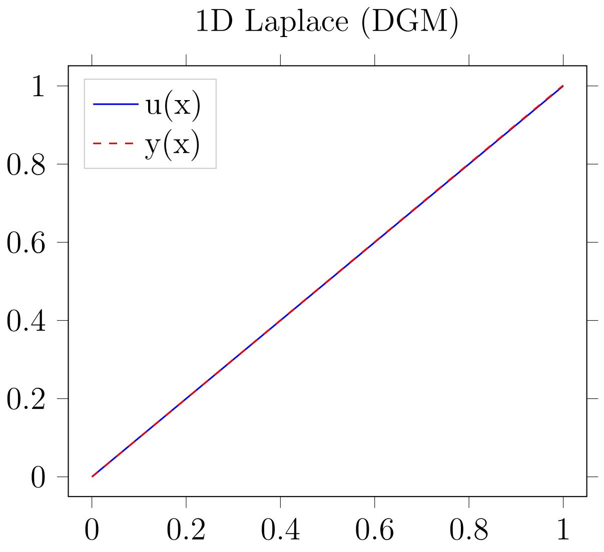

Figure 2.1: The dashed red line is our learned model, $y(\mathbf{w}, x)$, which is our approximation of the solution to 2.3. We use the tanh activation, five hidden layers each with 100 nodes, and train for 20 epochs; we use $\lambda = 1$. The blue line is the true solution. The model and true solution are imperceptibly different as plotted, with MAE of 18.6684×10^{-4}.

Layers	λ	Epochs	Tanh	Time	ReLU	Time
1	1	10	4.9440	7.76	123.4708	7.50
5	1	10	1.5732	29.46	150.5158	24.60
1	$\sqrt{10}$	10	13.9220	1.75	150.7669	7.53
5	$\sqrt{10}$	10	4.6481	29.54	240.8975	24.66
1	1	20	1.9660	15.65	179.3355	14.99
5	1	20	18.6684	59.14	183.1167	49.14
1	$\sqrt{10}$	20	4.9223	15.59	151.0015	15.19
5	$\sqrt{10}$	20	58.0302	58.89	143.1349	49.22

Table 2.1: A table depicting the final MAE of DGM networks trained to solve (2.3) for a random sample of 100 points on the interval $(0, 1)$ and training time while varying the hyper–parameters of our model. All MAE's are written as 10^{-4}. For example, the first line of data is from a single hidden–layer network, $\lambda = 1$, with 10 training epochs; when using the tanh activation, the network produced an output with MAE of 4.9440×10^{-4} and took 7.76 seconds to train; when using the ReLU activation, the network produced an output with MAE of 123.4708×10^{-4} and took 7.50 seconds to train.

choose for future problems. It is also possible that a longer training period would result in a better approximation for the deeper networks. Another feature to notice, when $\lambda = \sqrt{10}$, the MAE is consistently worse than when $\lambda = 1$. As previously mentioned, this term is not necessary in DGM.

We also need to discuss the training time. We use Python's Timeit package to measure this. Times will vary significantly from machine to machine and even on the same machine because of other operations the computer is performing. Therefore, we must only look at these times relative to one another and only qualitatively. We conduct these experiments in sequential order with little time in between each experiment to minimize variation in background computations. Note that ReLU is generally slightly faster than tanh; ReLU was specifically designed with computation speed in mind and we observe this to be true [2]. When we increase the depth of the network, it takes longer to train. This is because of the increase in number of parameters that must be trained. Lastly, doubling the number of training epochs roughly doubled the training time in each experiment. This comes as no surprise since the number of computations done is roughly doubled.

Unfortunately, trial–and–error is often the best way to adjust the hyper–parameters in a neural network, both for model accuracy and speed. The "black–box" nature of neural networks is still not well understood. This can be a serious issue in problems we do not have an explicit answer for. Just because the network gives you an output, does not mean it is correct; the network will give an answer even if a true solution does not exist. Neural networks also do not give any sort of confidence interval of a solution; although this is an area of active research [19]. With these pit–falls in mind, let us continue to a two–dimensional Laplace equation.

2.1.2 In Two Dimensions

Consider the following two–dimensional Laplace equation:

$$\begin{cases} \Delta u(r,\theta) = 0, \quad r \in (2,4), \quad \theta \in [0,2\pi) \\ u(2,\theta) = 0 \\ u(4,\theta) = 4\sin(5\theta) \end{cases} \tag{2.5}$$

Before we use DGM on this model, we can first solve for $u(r,\theta)$ explicitly so we can check the accuracy of the model.

$$\Delta u = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} = 0. \tag{2.6}$$

This is a separable equation so $u(r,\theta) = F(r)G(\theta)$ [13]. Therefore, after multiplying by r^2 and adding the second term to both sides, (2.6) becomes

$$\frac{r}{F}\frac{d}{dr}\left(r\frac{dF}{dr}\right) = -\frac{1}{G}\frac{d^2G}{d\theta^2} = n^2.$$

Since the left and right side are equal, but depend on different variables, they must be equal to some constant n^2. We then separate the equation into two ordinary differential equations:

$$r\frac{d}{dr}\left(r\frac{dF}{dr}\right) - n^2F = 0 \tag{2.7}$$

$$\frac{d^2G}{d\theta^2} + n^2G = 0. \tag{2.8}$$

Solving (2.7) and (2.8) we get the two general solutions:

$$F(r) = \begin{cases} A_n r^n + B_n r^{-n} & n \neq 0 \\ A_0 + B_0 \ln(r) & n = 0, \end{cases} \tag{2.9}$$

$$G(\theta) = C_n \cos(n\theta) + D_n \sin(n\theta). \tag{2.10}$$

Note that when $n = 0$, (2.9) becomes $F(r) = A_0 + B_0 \ln(r)$. By the inner boundary condition, $u(2, \theta) = 0$, we have $F(2) = 0 = A_0 + B_0 \ln(2)$. Therefore $A_0 = -B_0 \ln(2)$. By the outer boundary condition, $u(4, \theta) = 4\sin(5\theta)$, we would have $F(4) = 4\sin(5\theta)$. However, $F(r)$ cannot depend on θ, so $n \neq 0$. By the superposition principle [13, pg 17]:

$$u(r, \theta) = \sum_{n=1}^{\infty} (A_n r^n + B_n r^{-n})(C_n \cos(n\theta) + D_n \sin(n\theta)).$$

Multiplying out the terms in the sum leaves us with

$$u(r, \theta) = \sum_{n=1}^{\infty} (A_n r^n + B_n r^{-n}) \cos(n\theta) + \sum_{n=1}^{\infty} (C_n r^n + D_n r^{-n}) \sin(n\theta).$$

To satisfy the outer boundary condition, $u(4, \theta) = 4\sin(5\theta)$, we see $A_n = B_n = 0$ since any $\cos(n\theta)$ will never equal $\sin(n\theta)$ for the entire domain $[0, 2\pi)$. We also see the only possible n that will satisfy this condition is $n = 5$, since any other n will never equal $\sin(5\theta)$ on the entire interval $[0, 2\pi)$. So we are left with

$$u(r, \theta) = (C_5 r^5 + D_5 r^{-5}) \sin(5\theta).$$

We use our boundary conditions again to find C_5 and D_5.

$$u(2, \theta) = 0 = (C_5 (2)^5 + D_5 (2)^{-5}) \sin(n\theta).$$

Since this must hold for all $\theta \in [0, 2\pi)$, we must have

$$0 = (C_5(2)^5 + D_5(2)^{-5}). \tag{2.11}$$

Using our outer boundary condition, we have

$$u(4, \theta) = 4\sin(5\theta) = (C_5(4)^5 + D_5(4)^{-5})\sin(5\theta).$$

Observe that the $\sin(5\theta)$ terms cancel and we are left with

$$4 = C_5(4)^5 + D_5(4)^{-5}. \tag{2.12}$$

Using (2.11) and (2.12), we find $C_5 = \dfrac{4}{1023}$ and $D_5 = -\dfrac{4096}{1023}$. Thus the solution to (2.3) is

$$u(r, \theta) = \left(\frac{4}{1023}r^5 - \frac{4096}{1023}r^{-5}\right)\sin(5\theta). \tag{2.13}$$

Now that we have a solution, we create several neural networks with different hyper–parameters and train them to solve (2.5). We use (2.13) as the ground truth for measuring the accuracy of our model.

We now construct the loss function to train our network on. Our loss function for this problem is similar to the one–dimensional example; however, we must now sample multiple points from the boundary. Let $(2, \theta_i)$ be a randomly selected point on the inner boundary of the annulus, $(4, \theta_j)$ be a randomly selected point on the outer boundary of the annulus, and (r_k, θ_k) be a randomly selected point on the interior of the annulus. Note we again have a λ term here[1]. If we randomly sample M points from both the outer and inner boundary of the annulus, and N points from

[1]This is not necessary, as is seen in Table (2.1.2); again, initial experiments suggested increasing the boundary weight in the loss function yielded better results.

the interior of the annulus, our loss becomes:

$$\begin{aligned} E = \frac{\lambda}{M}\sum_{i=1}^{M}|y(\mathbf{w},2,\theta_i)-0|^2 + \frac{\lambda}{M}\sum_{j=1}^{M}|y(\mathbf{w},4,\theta_j)-4\sin(5\theta_j)|^2 \\ + \frac{1}{N}\sum_{k=1}^{N}|\Delta y(\mathbf{w},r_k,\theta_k)-0|^2 \end{aligned} \tag{2.14}$$

Note that computationally, we calculate $\Delta y(\mathbf{w}, r_k, \theta_k)$ using rectangular coordinates. We can see a "well trained" model in Figure 2.2, and data collected from a variety of networks with varying hyper–parameters in Table 2.1.2. We see that using SeLU or tanh as the activation has little effect on how well the model trains. Using SeLU seems to increase training times slightly over tanh. Using three hidden–layers seems to yield better results for this model than a single layer, or five layers, but this could again be due to a low number of training epochs. Penalizing the model for having large magnitude weights does not seem to have affected the accuracy of the model. This penalization is called "regularization" and is accomplished by adding $\|\mathbf{w}\|^2$ to our loss function. Some of these models clearly need to be trained longer to achieve their optimal solution. In particular, many of them had not sufficiently satisfied the interior boundary condition yet. This issue is seen even in Figure 2.2 with the longest training period.

2.1.3 Discussion of DGM

DGM can be very effective at solving certain PDE's. However, it is still in its infancy, and is in current development [7, 9]. For the Laplace equation, we need to calculate the second derivative of each of our sampled interior points. This requires the network to perform automatic differentiation twice, which takes roughly twice as long. In a higher order equation, this linear increase in computation time could be inefficient, but can be improved using a number of methods. The solution to the

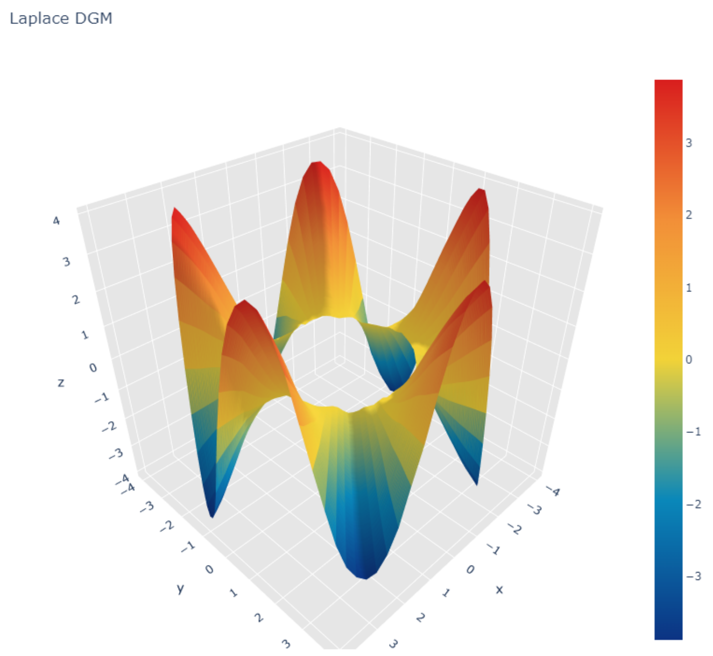

Figure 2.2: Model trained to solve (2.5). This model is trained using five hidden layers each with 100 nodes, the SeLU activation function[3], and 100 training epochs. $\lambda = 1$ and the final MAE is 4.1968×10^{-1}.

Layers	λ	Epochs	Tanh	Time	SeLU	Time
5	1	20	6.3167	133.64	4.7894	138.64
5	10	20	5.4824	133.56	4.6805	140.32
3	1	20	5.0687	75.43	4.8698	78.45
3	10	20	6.7066	75.16	5.2035	79.10
1	1	20	8.3804	36.44	10.1832	37.39
1	100	20	9.0161	37.06	8.4458	37.87
1	1	100	9.5690	181.64	8.6276	186.26
3	1	100	5.2778	373.34	6.2364	387.25
5	1	100	3.9714	567.91	4.1968	593.84
5 Regularized	1	20	6.3908	132.71	5.6889	142.35

Table 2.2: A table depicting the final MAE of multiple DGM networks trained to solve (2.5) from a random sample of 100 points on the annulus, with outer radius 4 and inner radius 2, and training time while varying the hyper–parameters of our model. All MAE's are written as 10^{-1}. For example, the first line of data is from a five hidden–layer network with 20 training epochs; when using the tanh activation, the network produced an output with MAE of 6.3167×10^{-1} and took 133.64 seconds to train; when using the SeLU activation, the network produced an output with MAE of 5.0085×10^{-1} and took 159.40 seconds to train.

Laplace equation, in particular, can sometimes be written as the minimizer to the Dirichlet functional [22], which has only a single gradient in its formula. Perhaps we can use this to increase the efficiency of training our model.

CHAPTER THREE

DEEP VARIATIONAL METHODS

Occasionally, PDEs, such as the Laplace Equation with special conditions, have a so–called "variational formulation". This variational formulation is a "functional", which maps functions to the real numbers. In these cases the solution to a PDE is the minimizer to the functional. From a computational standpoint, in the context of deep learning, the variational problem may be less expensive to solve, since this formulation tends to have fewer derivatives to compute. It is special that PDE's have an equivalent variational formulation, but there is a large class of functional minimization problems that our proposed method will work for.

Using these ideas, we create a new family of deep learning techniques to solve these variational problems. With inspiration from DGM, we train neural networks to satisfy the boundary conditions, initial conditions, etc; however, these networks minimize a functioanl, rather than satisfy a differential operator from the PDE formulation. The variational formulation often has lower order derivatives than the equivalent PDE problem, so in the rare case that a PDE has a variational formulation, this method would require less automatic differentiation than DGM.

3.1 Calculus of Variations

Elementary calculus is the study of small changes in the values of functions. Calculus of variations is the study of small changes in functions themselves. These are called variations [6]. We motivate this with an example. Consider an integrand, $f[x, u(x), u'(x)]$, assume it is real–valued. The definite integral of this integrand maps the function $u(x)$ to a real number. This integral is called a functional. The class of functionals we consider are definite integrals. Many problems arise where our integrand contains an unknown function and its derivatives. In these problems, we desire to find the function, $u(x)$, that minimizes the integral.

Consider an unknown function $u(x)$ on the interval $[0, 1]$ where $u(0) = 0$ and $u(1) = 1$. What function $u(x)$ satisfies these conditions has the smallest arc length? Intuitively, we know that the solution is $u(x) = x$, since a straight line will form the shortest path between two points. However, we can show this much more rigorously. Recall that the functional describing the arclength of a function $u(x)$ on the interval $[0, 1]$ is

$$S(u(x)) = \int_0^1 \sqrt{1 + (u'(x))^2}\, dx. \tag{3.1}$$

3.1.1 Functional Derivative

We must first discuss what it means for a functional to be optimal. In elementary calculus, unconstrained optimizers must be found where our derivative vanishes; these are known as "stationary points". The derivative of a function $u(x)$ is defined as:

$$\frac{du}{dx} = \lim_{\epsilon \to 0} \frac{u(x + \epsilon) - u(x)}{\epsilon},$$

where ϵ is some small change in our argument x. We have an analogous definition for functionals. Consider the following functional $S[u]$:

$$S[u] = \int_{x_1}^{x_2} f(u(x))\, dx,$$

where f is a function of some $u(x)$; $u(x_1) = u_1$ and $u(x_2) = u_2$. We can analyze this functional by observing how it changes with small changes to u. We can do this rigorously by adding a "small" perturbation function, $\epsilon\eta(x)$, $\epsilon > 0$, to $u(x)$. We enforce $\eta(x_1) = \eta(x_2) = 0$ since our boundary conditions must be satisfied. We then take the limit as ϵ goes to 0 and observe the effects on our functional. Our definition of our directional derivative of S with respect to u in the direction η then becomes:

$$\int_{x_1}^{x_2} \frac{\delta S}{\delta u}\eta(x)\,dx = \lim_{\epsilon\to 0}\frac{S[u+\epsilon\eta]-S[u]}{\epsilon} = \left[\frac{d}{d\epsilon}S[u+\epsilon\eta]\right]_{\epsilon=0}$$

Now that we have a definition of the functional derivative we can begin searching for optimizers. Just as in elementary calculus, we must find the “stationary points”. That is, the u’s where our functional derivative vanishes [16, pg 217–218]. Note, just as in elementary calculus, this is not enough to guarantee a maximum or minimum [16, pg 217–218].

3.1.2 Derivation of the Euler–Lagrange Equation

We now derive the Euler–Lagrange equation whose solutions are stationary points of a given functional [16]. We first restate our requirement that our problem be well posed. Functionals, even if bounded, may not have a minimizer or even a stationary point. Consider the following functional which we wish to find the stationary points, u, of; assume they exist:

$$S[u] = \int_{x_1}^{x_2} f(x, u(x), u'(x))\,dx. \tag{3.2}$$

This functional is dependent on x, an unknown function u and its derivative u'. We have the constraint that $u(x_1) = u_1$ and $u(x_2) = u_2$. We find u that satisfies these conditions and minimizes the functional $S[u]$. Assume u is a stationary point of S. Call any “neighboring function” $U(x;\epsilon) = u(x) + \epsilon\eta(x)$, where $\eta(x_1) = \eta(x_2) = 0$ and $\epsilon > 0$. This point, $U(x;\epsilon)$, may not be stationary.

$$S[U(x;\epsilon)] = \int_{x_1}^{x_2} f(x, u(x)+\epsilon\eta(x), u'(x)+\epsilon\eta'(x))\,dx = \int_{x_1}^{x_2} f(x, U(x;\epsilon), U'(x;\epsilon))\,dx.$$

We want to observe how S changes with changing ϵ, so we differentiate with respect to ϵ:

$$\frac{dS}{d\epsilon} = \frac{d}{d\epsilon}\int_{x_1}^{x_2} f[x, U(x;\epsilon), U'(x;\epsilon)]\, dx.$$

If we assume $f(x, U(x;\epsilon), U'(x;\epsilon))$ and $\dfrac{\partial}{\partial\epsilon} f(x, U(x;\epsilon), U'(x;\epsilon))$ are continuous in x and ϵ, then by the Leibniz integral rule we can change the order of differentiation and integration and are left with

$$\int_{x_1}^{x_2} \frac{\partial}{\partial\epsilon} f(x, U(x;\epsilon), U'(x;\epsilon))\, dx.$$

Using the chain rule we get

$$\int_{x_1}^{x_2} \frac{\partial f}{\partial x}\frac{\partial x}{\partial \epsilon} + \frac{\partial f}{\partial U}\frac{\partial U}{\partial \epsilon} + \frac{\partial f}{\partial U'}\frac{\partial U'}{\partial \epsilon}\, dx = \int_{x_1}^{x_2} \frac{\partial f}{\partial U}\eta(x) + \frac{\partial f}{\partial U'}\eta'(x)\, dx. \tag{3.3}$$

Applying integration by parts to the second term we are left with

$$\int_{x_1}^{x_2} \eta'(x)\frac{\partial f}{\partial U'}\, dx = -\int_{x_1}^{x_2} \eta(x)\frac{d}{dx}\left[\frac{\partial f}{\partial U'}\right] dx,$$

since $\eta(x_1) = \eta(x_2) = 0$. We can substitute this back into (3.3) and are left with

$$\frac{dS}{d\epsilon} = \int_{x_1}^{x_2} \eta(x)\frac{\partial f}{\partial U} - \eta(x)\frac{d}{dx}\left[\frac{\partial f}{\partial U'}\right] dx = \int_{x_1}^{x_2} \eta(x)\left(\frac{\partial f}{\partial U} - \frac{d}{dx}\left[\frac{\partial f}{\partial U'}\right]\right) dx.$$

If $\epsilon = 0$, we have $U(x;0) = u(x)$ and for $S[u]$ to be stationary we therefore must have that

$$0 = \frac{dS}{d\epsilon} = \int_{x_1}^{x_2} \eta(x)\left(\frac{\partial f}{\partial u} - \frac{d}{dx}\left[\frac{\partial f}{\partial u'}\right]\right) dx. \tag{3.4}$$

Now since η is an arbitrary function, this means we require

$$\frac{\partial f}{\partial u} - \frac{d}{dx}\left[\frac{\partial f}{\partial u'}\right] = 0. \tag{3.5}$$

This is known as the Euler–Lagrange equation [16, pg 218–220] and we can use this to the stationary points of (3.2)[1]. The jump from (3.4) to (3.5) is a result of the "fundamental lemma of calculus of variations" which we do not state. However, we can see this holds if we assume $\frac{\partial f}{\partial u} - \frac{d}{dx}\left[\frac{\partial f}{\partial u'}\right]$, denote it as $g(x)$, is continuous on the interval $[x_1, x_2]$. Consider the case that $g(x)$ is zero everywhere except at a single point between x_1 and x_2. Then the integral, $\int_{x_1}^{x_2} \eta(x)g(x)$, is still zero with an arbitrary $\eta(x)$, but $g(x)$ is non–zero on $[x_1, x_2]$.

Now that we can find the stationary points of a functional dependent on some $u(x)$ and $u'(x)$, assuming such stationary points exists, we can finally solve (3.1).

$$S[u] = \int_0^1 \sqrt{1 + (u'(x))^2}\, dx.$$

For S to be minimal, it must have a stationary point, $u(x)$, satisfying (3.5). We note that $f(x, u(x), u'(x)) = \sqrt{1 + (u'(x))^2}$. Thus we have that

$$\frac{\partial f}{\partial u} = 0,$$

since f does not depend on $u(x)$. Therefore, (3.5) becomes

$$\frac{d}{dx}\frac{\partial f}{\partial u'} = 0.$$

[1]This formula can be generalized to higher dimensions and with higher derivatives.

Taking the partial derivative of f with respect to $u'(x)$ we are left with

$$\frac{d}{dx}\frac{u'(x)}{\sqrt{1+(u'(x))^2}} = 0.$$

Since the derivative of this function with respect to x is 0, we must have that

$$\frac{u'(x)}{\sqrt{1+u'^2}} = c,$$

for some $c \in \mathbb{R}$. Solving for $u'(x)$ we get

$$u'(x) = \pm\sqrt{\frac{c^2}{1-c^2}}.$$

The right hand side is just an unknown constant, call it m. Then we are left with the ordinary differential equation

$$u'(x) = m. \tag{3.6}$$

Integrating both sides with respect to x gives us

$$u(x) = mx + b.$$

Finally, applying our boundary conditions, $u(0) = 0$ and $u(1) = 1$, we find $m = 1$ and $b = 0$. Thus, we are left with the solution which we intuitively know: $u(x) = x$.

Notice that applying (3.5) to our functional (3.1) turned the problem into the differential equation (3.6). This shows the minimizer of the functional (3.1) is the solution to (3.6). In essence, the two problems are equivalent; we will explore this equivalence more in a later example. Computationally, these problems are different, however. So can we modify our DGM model to instead find the minimizer to a functional rather than solve a differential equation?

3.2 Deep Variational Methods

Deep variational methods (DVMs) are a family of deep learning algorithms we propose to find minimizers to a given functional with boundary conditions. This novel approach, in certain circumstances, can be more efficient than performing DGM on the equivalent differential equation problem. Let us explore how. Consider the following functional optimization problem:

$$\begin{cases} \min\limits_{u \in Q} & J(u(\mathbf{x})), \quad \mathbf{x} \in \Omega \\ \text{subject to} & u(\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega. \end{cases} \tag{3.7}$$

Where Q is some set of allowed functions, $J(u(\mathbf{x})) = \int_\Omega f(\mathbf{x}, u(\mathbf{x}), \nabla u(\mathbf{x}), \cdots)\, d\mathbf{x}$, $\Omega \subseteq \mathbb{R}^D$ is an open set and $\mathbf{x} \in \mathbb{R}^D$. First, we assume $J(u(\mathbf{x}))$ is bounded from below; then we assume there exists a $u(\mathbf{x})$ such that the boundary conditions are satisfied and $J(u(\mathbf{x}))$ is minimal. If these conditions are not met, DVM will fail. These conditions can be verified using various methods.

While in DGM, we train a neural network to satisfy the differential operator and boundary conditions, in DVM, we train a neural network to satisfy boundary conditions and to minimize the functional. However, calculating the integral at every step would be costly, so we estimate it by taking a sample of N points from Ω, calculating $f(\mathbf{x}_i, u(\mathbf{x}_i), \nabla u(\mathbf{x}_i), \cdots)$ at each of these points, sum up these values, and then normalizing by the number of points we select[2]. We then update our network parameters to minimize this sum. The approximation can be succinctly written as:

$$\int_\Omega f[\mathbf{x}, u(\mathbf{x}), \nabla u(\mathbf{x}), \cdots]\, d\mathbf{x} \approx \frac{1}{N} \int_\Omega d\mathbf{x} \sum_{i=1}^{N} f[\mathbf{x}_i, u(\mathbf{x}_i), \nabla u(\mathbf{x}_i), \cdots]. \tag{3.8}$$

[2]This technique for estimating our integral is called Monte Carlo integration [8].

Now we must also train the network to satisfy the boundary conditions. As in DGM, we select random points from the boundary, $\partial\Omega$, and penalize the network for not satisfying $u(\mathbf{x_i}) = g(\mathbf{x_i})$. Now we can construct a loss function to train our neural network on. Assume we randomly sample M points Ω and N points from $\partial\Omega$. Again, let $y(\mathbf{w}, \mathbf{x})$ be the network's approximation for $u(\mathbf{x})$. We now arrive at the following loss function:

$$E = \frac{1}{M}\sum_{i=1}^{M} f(\mathbf{x}_i, y(\mathbf{w}, \mathbf{x}_i), \nabla y(\mathbf{w}, \mathbf{x}_i), \cdots) + \lambda \frac{1}{N}\sum_{j=1}^{N} |y(\mathbf{w}, \mathbf{x}_j) - g(\mathbf{x}_j)|^2. \tag{3.9}$$

Note that we again have a λ with our boundary condition term. In DVM, this term is required. Recall that in DGM, the network is trained to satisfy the differential operator for all points in Ω directly. In DVM, we do not train the network to satisfy a differential operator; we train it minimize an integral. At the minimum, this functional is typically non–zero; thus our loss function evaluated at the true solution will be non–zero. This is not the case in DGM. DVM is known as a "penalty method" [5, pg 564]. In a penalty method, we approximate a constrained minimization problem as an unconstrained problem. Consider the following constrained optimization problem:

$$\begin{cases} \min\limits_{\mathbf{x}} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{x} \in \Omega \subseteq \mathbb{R}^D \end{cases} \tag{3.10}$$

We can approximate (3.10) by the following unconstrained problem:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) + \lambda P(\mathbf{x}), \tag{3.11}$$

where $\lambda > 0$ is a "penalty parameter", and $P : \mathbb{R}^D \to \mathbb{R}$ is a "penalty function" [5, pg 564–565]. A penalty function, P, must be continuous; $P(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{R}^D$; $P(\mathbf{x}) = 0$ if and only if $\mathbf{x} \in \Omega$.

In DVM, our penalty function is the L_2 error between our model, $y(\mathbf{w}, \mathbf{x})$, and $g(\mathbf{x})$ for all $\mathbf{x} \in \partial\Omega$. We approximate this L_2 error as the MSE of a random sample of points in $\partial\Omega$. See the second sum in (3.9). This function is continuous, always non–negative, and zero if and only if $y(\mathbf{w}, \mathbf{x}) = g(\mathbf{x})$ for all $\mathbf{x} \in \partial\Omega$. If a chosen λ is not penalizing the model enough to force it to satisfy the boundary conditions, we increase λ; this is done by hand. We conjecture without proof that: $y(\mathbf{w}, \mathbf{x}) - g(\mathbf{x}) = O({}^1\!/\!_\lambda)$ as $\lambda \to \infty$ for all $\mathbf{x} \in \partial\Omega$. Thus, ideally, we could pick $\lambda = \infty$; however, this is computationally infeasible. For now, we pick different finite λ's and train the network using each of them.

Since we use a penalty method, we are approximating the constrained minimization problem as a, so can this method really be as effective as DGM? We now move into some examples.

3.3 The Laplace Equation Revisited

Let $\Omega \subset \mathbb{R}^D$, and denote $\overline{\Omega} = \Omega \cup \partial\Omega$. Now, consider the following problem:

$$\begin{cases} \Delta u(\mathbf{x}) = 0, & \mathbf{x} \in \Omega \\ u(\mathbf{x}) = f(\mathbf{x}), & \mathbf{x} \in \partial\Omega \end{cases} \tag{3.12}$$

Using the Dirichlet Principle, we wish to replace (3.12) by the following minimization problem:

$$I = \inf_v \left\{ \int_\Omega |\nabla v(\mathbf{x})|^2 \, d\mathbf{x} : v(\mathbf{x}) \in C^2(\overline{\Omega}) \text{ and } v(\mathbf{x}) = f(\mathbf{x}) \text{ on } \partial\Omega \right\} \tag{3.13}$$

We say $v(\mathbf{x})$ is "admissible" if $v(\mathbf{x}) \in C^2(\overline{\Omega})$ and $v(\mathbf{x}) = f(\mathbf{x})$ on $\partial\Omega$ [11].

Theorem 3.3.1. Given $u(\mathbf{x})$ is admissible, then $u(\mathbf{x})$ is a solution to (3.12) if and only if u minimizes (3.13) [11].

Proof. ($\Rightarrow$) Let $u(\mathbf{x})$ be admissible, and assume $u(\mathbf{x})$ is a solution to (3.12). To show that $u(\mathbf{x})$ is a minimizer to (3.13), we must show that for any admissible function $v(\mathbf{x})$:

$$\int_\Omega |\nabla u(\mathbf{x})|^2 \, d\mathbf{x} \leq \int_\Omega |\nabla v(\mathbf{x})|^2 \, d\mathbf{x}. \tag{3.14}$$

First, we show a result of Green's Identity which states [13, pg 15]

$$\int_\Omega f(\mathbf{x})\Delta g(\mathbf{x}) \, d\mathbf{x} = \int_{\partial\Omega} f(\mathbf{x})\nabla g(\mathbf{x}) \cdot \hat{\mathbf{n}} \, d\mathbf{x} - \int_\Omega \nabla f(\mathbf{x}) \cdot \nabla g(\mathbf{x}) \, d\mathbf{x}.$$

Rearranging the terms we get

$$\int_\Omega \nabla f(\mathbf{x}) \cdot \nabla g(\mathbf{x}) \, d\mathbf{x} = \int_{\partial\Omega} f(\mathbf{x})\nabla g(\mathbf{x}) \cdot \hat{\mathbf{n}} \, d\mathbf{x} - \int_\Omega f(\mathbf{x})\Delta g(\mathbf{x}) \, d\mathbf{x}.$$

Replacing $f(\mathbf{x})$ with $v(\mathbf{x}) - u(\mathbf{x})$ and $g(\mathbf{x})$ with $u(\mathbf{x})$ we get

$$\begin{aligned}\int_\Omega \nabla(v(\mathbf{x}) - u(\mathbf{x})) \cdot \nabla u(\mathbf{x}) \, d\mathbf{x} = &\int_{\partial\Omega} (v(\mathbf{x}) - u(\mathbf{x}))\nabla u(\mathbf{x}) \cdot \hat{\mathbf{n}} \, d\mathbf{x} \\ &- \int_\Omega (v(\mathbf{x}) - u(\mathbf{x}))\Delta u(\mathbf{x}) \, d\mathbf{x}.\end{aligned}$$

However, $v(\mathbf{x}) = u(\mathbf{x})$ on $\partial\Omega$ since both are admissible functions; thus, the first term on the right becomes zero. We also note that since $u(\mathbf{x})$ is a solution to (3.12), $\Delta u(\mathbf{x}) = 0$ on Ω; thus, we are left with

$$\int_\Omega \nabla(u(\mathbf{x}) - v(\mathbf{x})) \cdot \nabla u(\mathbf{x}) \, d\mathbf{x} = 0. \tag{3.15}$$

Now consider

$$\begin{aligned}\int_\Omega |\nabla v(\mathbf{x})|^2\,d\mathbf{x} &= \int_\Omega |\nabla(u(\mathbf{x}) + (v(\mathbf{x}) - u(\mathbf{x})))|^2\,d\mathbf{x} \\ &= \int_\Omega |\nabla u(\mathbf{x})|^2\,d\mathbf{x} + 2\int_\Omega \nabla u(\mathbf{x})\cdot\nabla(v(\mathbf{x}) - u(\mathbf{x}))\,d\mathbf{x} \\ &\quad + \int_\Omega \nabla|(v(\mathbf{x}) - u(\mathbf{x}))|^2\,d\mathbf{x}.\end{aligned}$$

By (3.15), we have

$$\begin{aligned}&= \int_\Omega |\nabla u(\mathbf{x})|^2\,d\mathbf{x} + \int_\Omega |\nabla(v(\mathbf{x}) - u(\mathbf{x}))|^2\,d\mathbf{x} \\ &\geq \int_\Omega |\nabla u(\mathbf{x})|^2\,d\mathbf{x}.\end{aligned}$$

Which is precisely (3.14) as desired.

($\Leftarrow$) Let $u(\mathbf{x})$ be admissible, and assume $u(\mathbf{x})$ minimizes (3.13). We must show that $\Delta u(\mathbf{x}) = 0$ in Ω. Let $\phi(\mathbf{x}) \in C_0^\infty(\Omega)$ be a smooth functions with compact support on Ω). Let

$$F(t) = \int_\Omega |\nabla(u(\mathbf{x}) + t\phi(\mathbf{x}))|^2\,d\mathbf{x} - \int_\Omega |\nabla u(\mathbf{x})|^2\,d\mathbf{x} \;\; \forall\, t \in \mathbb{R}.$$

Since u is a minimizer of the integral, $F(t) \geq 0$ and $F(0) = 0$. Thus, $F'(0) = 0$. Observe that

$$F'(t) = 2\int_\Omega \nabla u(\mathbf{x})\cdot\nabla\phi(\mathbf{x})\,d\mathbf{x} + 2t\int_\Omega |\nabla\phi(\mathbf{x})|^2\,d\mathbf{x}.$$

We now see that

$$0 = F'(0) = 2\int_\Omega \nabla u(\mathbf{x})\cdot\nabla\phi(\mathbf{x})\,d\mathbf{x} = -2\int_\Omega \Delta u(\mathbf{x})\phi(\mathbf{x})\,d\mathbf{x}.$$

However, $\phi(\mathbf{x})$ was arbitrary, so we must have $\Delta u(\mathbf{x}) = 0$ in Ω. ■

We have shown that the minimizer to (3.13) is also a solution to (3.12) analytically, now. However, we reiterate the existence of such a minimizer is rare.

3.3.1 In One Dimension

Consider the following one–dimensional functional optimization problem:

$$\begin{cases} \min\limits_{u} & \displaystyle\int_0^1 |\nabla u(x)|^2 \, dx \\ \text{subject to} & u(0) = 0 \\ & u(1) = 1 \end{cases} \tag{3.16}$$

where $u(x)$ is an admissible function. By the Dirichlet principle, this is equivalent to the one–dimensional problem, (2.3). Now let us see how DVM is performed. We take a uniform random sample of N points from the interval $(0,1)$. Since the boundary consists of just two points, we do not randomly sample points from the boundary; we simply apply our λ to the boundary conditions as previously discussed. So our loss function for this becomes

$$E = \lambda(|y(\mathbf{w},0) - 0|^2 + |y(\mathbf{w},1) - 1|^2) + \sum_{i=1}^{N} \|\nabla y(\mathbf{w}, x_i)\|^2 \tag{3.17}$$

We now create and train several neural networks with varying hyper–parameters to find the $y(\mathbf{w}, x)$ that minimizes (3.17). In each training epoch we use 100 iterations of the algorithm, and in each iteration we choose 100 random points from the interval $(0,1)$. We also use 100 nodes in each hidden layer as we did in our DGM examples. Results of these experiments can be seen in Table 3.3.1.

As with the DGM approach, networks using the tanh activation generally outperformed the networks using the ReLU activation. This is again likely caused by the fact that ReLU is not a smooth function. Also note that increasing λ generally

Layers	λ	Epochs	Tanh	Time	ReLU	Time
1	1	10	234.7055	7.00	258.9457	6.94
1	10	10	167.3706	6.00	178.9885	7.07
1	100	10	5.1360	6.82	5.4397	6.93
1	1000	10	1.7596	7.01	1.2487	6.95
5	1000	10	14.6392	20.85	36.0383	20.60
5	1000	20	18.9654	41.35	1.8704	41.22
1	1000	20	2.5499	13.83	23.3278	13.85
1	10000	20	3.5313	13.75	7.48198	13.83

Table 3.1: A table depicting the final MAE of trained DVM neural networks for a random sample of 100 points on the interval $(0, 1)$ and training time while varying the hyper–parameters of our model. All MAE's are written as 10^{-3}. For example, the first line of data is from a single hidden–layer network, $\lambda = 1$, and with 10 training epochs; when using the tanh activation, the network produced an output with MAE of 234.7055×10^{-3} and took 7.00 seconds to train; when using the ReLU activation, the network produced an output with MAE of 258.9457×10^{-3} and took 6.94 seconds to train.

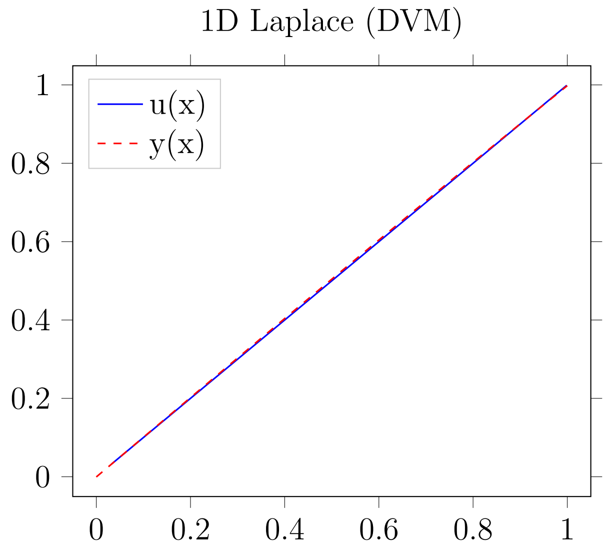

Figure 3.1: A visualization of a DVM networks trained to solve 3.16. Using one hidden layer, $\lambda = 1000$, the tanh activation, and 20 training epochs. The final MAE of the network is 2.5499×10^{-3}.

increased the accuracy of the model. Since we conjecture $\lambda = \infty$ would be ideal as the network approaches the solution, this trend is expected. A visualization of some of these results can be seen in Figure 3.3.1 and 3.3.1.

Now that we have seen DVM work, we can compare this method to the DGM approach for the same problem. Our first observation is DGM is about one order of magnitude more accurate than DVM in most examples. This is due to the nature of penalty methods: namely the trade–off between boundary conditions and the functional objective. The next thing to consider is training time. Recall both methods use automatic differentiation at each one of our sampled training points. For DVM, we only need to calculate the gradient, not the Laplacian at each of these points, so we would expect DVM to be faster. This is visible in our results. In examples where the number of hidden layers and training epochs are the same, DVM is faster

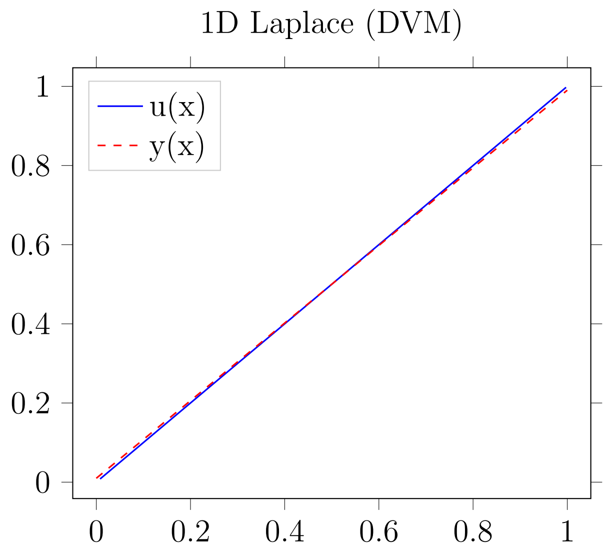

Figure 3.2: A visualization of a DVM networks trained to solve 3.16. Using one hidden layer, $\lambda = 100$, the ReLU activation, and 10 training epochs. Notice the error is perceptible in this figure showing that ReLU is not the ideal activation for this application. The final MAE of the network is 5.4397×10^{-3}.

despite each network needing to learn the same number of parameters. This is the main benefit of DVM over DGM. In higher dimensional cases, we would expect this trend to continue. So let us look at the two–dimensional case.

3.3.2 In Two Dimensions

As before, we will look at the same problem we tackled using DGM, but using DVM techniques. Consider the following functional optimization problem:

$$\begin{cases} \min\limits_{u} & \int_0^{2\pi}\int_2^4 \|\nabla u(r,\theta)\|^2 \, dr \, d\theta \\ \text{subject to} & u(2,\theta) = 0 \\ & u(4,\theta) = 4\sin(5\theta) \end{cases} \tag{3.18}$$

where $u(r,\theta)$ is admissible. Similarly to our one–dimensional example, this problem is equivalent to (2.5) by the Dirichlet Principle. We create several DVM networks similar to those we created before and compare the results of DGM and DVM. We sample M points from each boundary and N points from the interior of the annulus. Modifying (3.9) for this probelm we arrive at the following loss function to minimize:

$$E = \frac{\lambda}{M}\sum_{i=1}^{M} |y(\mathbf{w}, 2, \theta_i) - 0|^2 + \frac{\lambda}{M}\sum_{j=1}^{M} |y(\mathbf{w}, 4, \theta_j) - 4\sin(5\theta_j)|^2 + \frac{\beta}{N}\sum_{k=1}^{N} |\nabla y(\mathbf{w}, \mathbf{x}_k)|^2 \tag{3.19}$$

The β term is not necessary just as λ was not necessary in the DGM model, but initial experiments showed it may help the model train. Similarly to our DGM model, we compute $\nabla y(\mathbf{w}, \mathbf{x_k})$ using rectangular coordinates.

In each iteration of each example we sample 100 points from the interior of the annulus and 100 points from each boundary. Each hidden layer has 100 nodes just as before. We can visualize the learned model in Figure 3.3. We can see the results of varying the hyper–parameters of our DVM networks in Table 3.3.2.

We again see that a larger λ produced a more accurate model in general. The use of tanh or SeLU seems to have little effect on model accuracy or training time. Regularization, as with the DGM model, has little influence on model accuracy.

Now the comparison of our DVM results with our DGM results is more interesting. The DGM and DVM models have very similar MAEs. This result is different form our one–dimensional cases and worthy of note since DGM was intended to be used in high–dimensional situations. So the next thing to consider is the training time. In most cases in which the number of hidden layers and number of training epochs are the same, DVM is substantially faster; in some cases DVM is twice as fast as DGM. Again, this is because for these Laplace problems, DVM must only calculate the gradient at the interior points, not the Laplacian. Automatic differentiation is an expensive process, and having to do it twice for each point slows down the training substantially.

Layers	β	λ	Epochs	Tanh	Time	SeLU	Time
5	0.001	1	20	5.1010	69.74	5.2591	66.42
5	0.01	1	20	2.8228	69.63	3.5809	67.93
5	0.1	1	20	5.5361	69.77	6.9833	67.29
5	1	1	20	8.9173	69.77	7.9236	67.75
5	1	10	20	8.7821	69.72	8.4319	66.43
5	1	100	20	3.4822	65.58	3.9432	67.43
5	1	1000	20	4.4597	65.33	4.7562	67.00
5	0.001	1000	20	5.4856	66.44	6.0659	67.26
5	1	1000	100	2.5649	323.78	3.9997	337.75
3	1	1000	20	5.4721	44.59	4.8872	46.09
5 Regularized	1	1000	20	4.8714	79.97	4.4376	80.23
5 Regularized	1	1000	100	2.7419	397.48	3.9956	409.56

Table 3.2: A table depicting the final MAE of several DVM neural networks for a random sample of 100 points on the annulus, with outer radius 4 and inner radius 2, and training time while varying the hyper–parameters of our model. All MAE's are written as 10^{-1}. For example, the first line of data is from a five hidden–layer network with $\beta = 0.001$, $\lambda = 1$, and trained for 20 training epochs; when using the tanh activation, the network produced an output with MAE of 5.1010×10^{-1} and took 69.74 seconds to train; when using the SeLU activation, the network produced an output with MAE of 5.2591×10^{-1} and took 66.42 seconds to train.

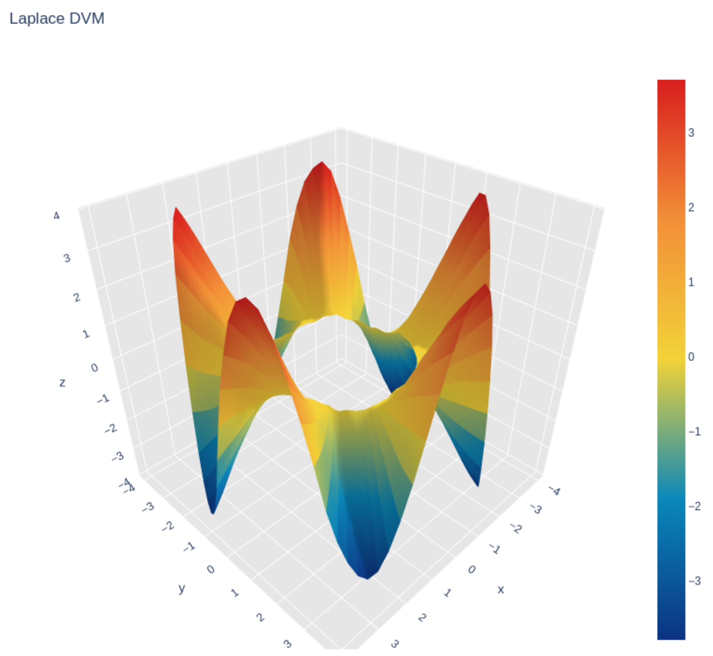

Figure 3.3: A visualization of a two dimensional DVM model trying to minimize (3.18). This model uses five hidden layers, the SeLU activation with regularization, $\lambda = 1000$, $\beta = \frac{1}{1000}$, and 100 training epochs. Note the boundaries have visible error.

CHAPTER FOUR

CONCLUSIONS

Neural networks are a very powerful tool. Their ability to uniformly approximate continuous functions as compositions and linear combinations of non–linear activation functions has a wide range of applications. More advanced architectures than the simple feed–forward networks we use can yield better results depending on the problem; however, the "optimal architecture" for a given problem is difficult to determine. This "black–box" nature of neural networks makes using them somewhat dubious without careful considerations.

Despite these drawbacks, we see that neural networks have a wide range of applications; in particular, they can be powerful tools for solving PDEs. Deep Galerkin Methods, and extensions there of, can be used to solve problems in high–dimension without the need for a large mesh, while still providing the accuracy needed. For a small class of higher order problems with a variational formulation, DGM can be inefficient.

We address this inefficiency by introducing Deep Variational Methods in which we train a neural network to minimize the equivalent variational formulation of the PDE. In these examples, we show DVM has similar accuracy to DGM with less training time. However, DVM can work for functional minimization problems that do not have an equivalent PDE representation. We could have demonstrated DVM on the examples we showed without comparing them to their equivalent problem in DGM; however, being able to replicate previous work with a new method is important; also DGM was the primary inspiration for DVM. More research is needed to determine the specific class of functional minimization problems that DVM can be applied to.

REFERENCES CITED

[1] Martin et al Abadi. Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[3] Leah Bar and Nir Sochen. Unsupervised deep learning algorithm for pde-based forward and inverse problems. 2019.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2009.

[5] Edwin K. P. Chong and Stanislaw H. Zak. *An Introduction to Optimization.* Wiley, 2013.

[6] R Courant and D Hilbert. *Methods of Mathematical Physics.* Interscience Publishers, 1953.

[7] G Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS), 2, 303-314*, 1989.

[8] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media.* PhD thesis, UC San Diego, September 2008.

[9] George Em Karniadakis, Ionnis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics–informed machine learning. 2021.

[10] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[11] Augusto C. Ponce. Topics on calculus of variations. 2006.

[12] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics Vol. 378*, 2019.

[13] Sandro Salsa. *Partial Differential Equations in Action: Third Edition.* Springer, 2016.

[14] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics Vol. 375*, 2018.

[15] Yutaro Tachibana, Matthew J. Graham, Nobuyuki Kawai, S. G. Djorgoski, Andrew J. Drake, Ashish A. Mahabal, and Daniel Stern. Deep modeling of quasar variability, 2020.

[16] John R. Taylor. *Classical Mechanics*. University Science Books, 2005.

[17] Keras Team. Keras documentation: Optimizers.

[18] Keras Team. Keras documentation: Writing a training loop from scratch, Apr 2020.

[19] Nikita Vemuri. Scoring confidence in neural networks, 2020.

[20] Eric W Weisstein. l^2–norm from mathworld–a wolfram web resource.

[21] Scott Zoldi. Explaining "black box" neural networks, 2019.

[22] Dominique Zosso and Braxton Osting. A primal-dual optimization strategy for elliptic partial differential equations. *The Quarterly of Applied Mathematics Vol. 79*, 2021.