AMSI VACATION RESEARCH SCHOLARSHIPS 2018-2019

Performance of Artificial Neural Networks on Small Structured Datasets

Daniel Condon
Supervised by Adel Rahmani
University of Technology Sydney

Vacation Research Scholarships are funded jointly by the Department of Education and Training and the Australian Mathematical Sciences Institute.

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

Australian Government
Department of Education and Training

Contents

Contents 2
Introduction 3
Tools 4
Neural Network 4
Feedforward 4
Backpropagation 5
Theoretical Motivation 6
Universal Approximation Theorem 6
Entity Embedding 6
Datasets 8
Dataset 1 8
Dataset 2 8
Dataset 3 8
Classification 9
Models 9
Statsmodels – dataset 1 9
Statsmodels – dataset 2 9
Statsmodels – dataset 3 10
Statsmodels - classification 10
Tensorflow 10
Comparison Algorithm 11
Results 11
Regression 11
Classification 12
Regression vs Classification 13
Regression 13
Classification 13
Embedding 13
Future Research 14
Code 14
References 14

Abstract

This project explores the performance of Artificial Neural Networks compared with traditional statistical methods in machine learning tasks involving small structured datasets. Synthetic datasets are created with 2-3 features which have either a linear and polynomial relationship to the target variable. These datasets are created for both regression and classification tasks.
We compare traditional statistical methods such as linear regression, polynomial regression, and logistic regression with a simple ANNs to determine whether or not ANNs can compete with these traditional methods on smaller structured datasets. Specifically, we look at which number of observations and level of stochastic noise ANNs no longer compete with these methods.
We also experiment with the technique of entity embedding for categorical features to determine whether or not they improve the accuracy of the ANN for small datasets.

We find that for regression tasks, a simple ANN is not able to compete with linear or polynomial regression on any of our synthetic datasets with less than 1000 observations. However, we do find that for classification tasks ANNs can compete with logistic regression down to as low as 100 observations in some cases. Entity embedding does not seem to improve these models by any significant factor, although there is some evidence to suggest that in more complex datasets with a higher level of noise they have the potential to improve the accuracy of the model.

Introduction

In the last 5- 10 years Artificial Neural Networks (ANNs) have become the gold standard in machine learning on large unstructured datasets. For almost all perceptual tasks involving unstructured data such as images and sound, ANNs are currently unbeaten, winning all major machine learning competitions such as ImageNet, and smaller competitions on the popular online competition website Kaggle.[1]

The motivation for this research project comes from the fact that there has been lots of research done on the capabilities of Neural Networks on large datasets and unstructured datasets, however, there has not been much research on the specific capabilities of neural networks on smaller, structured datasets. This is understandable, as we have been able to use traditional statistical methods on tabular data for a very long time, and these methods are very quick and very accurate – indeed it is only because these traditional methods failed on large unstructured datasets that neural networks become attractive.

However, there is some use in knowing if neural networks can perform equivalently to these methods on smaller structured datasets for several reasons. In industry, there may be a pipeline for data analysis that usually handles large datasets or unstructured datasets. If on occasion the data being fed into the pipeline is smaller and in tabular form it may be more convenient to use the same machine learning architecture (ANNs) rather than a kernel-based, or statistical approach.

Klambauer et al have shown that neural networks underperform on standard UCI machine learning datasets of less than 1000 observations when compared with state-of-the-art machine learning

[1] Chollet, F. (2017). [1]

models.[2] We take motivation from this work and aim to determine a specific lower bound on the size of the data in which Neural Networks become ineffective when compared to a traditional statistical benchmark. We know that ANNs are not as effective when faced with small structured datasets – but just how small can the dataset be before it breaks down?

In order to answer this question, we compare the performance of ANNs with traditional linear, polynomial and logistic regression. We choose this traditional statistical method rather than the state-of-the-art kernel or tree-based methods to provide a benchmark and give a general heuristic for when an ANN may break down.

Finally, we take motivation from recent articles from the University of San Francisco's fast.ai team[3] which explore the use of a recent technique called entity embedding in order to improve the performance of neural networks on structured data which contain categorical features.

Tools

For this project, we heavily relied upon the opensource machine learning framework Tensorflow, specifically the python deep learning library Keras which provides a high-level interface to Tensorflow (among other frameworks). We perform all of our computation on Google Collaboratory, a cloud-based notebook similar to Jupyter[4], which allows us to use Google's cloud computing resources, including graphical processing units (GPU) and the more recent, deep learning oriented tensor processing unit (TPU), which speeds up our computations significantly. For our linear, polynomial and logistic regression we use the python package Statsmodels.[5]

Neural Network

The basic neural network which we use in our models is built on an architecture of layers, connecting nodes (neurons) with edges (synapses) as a very loose representation of a brain. Each node outputs a continuous number, its activation. Connections between nodes are established through weights, biases and a non-linear activation function. This non-linearity is what allows the ANN to model any continuous function, and is what gives this method of machine learning its power.

The nodes in the input layer are the features in the dataset, and the node in the output layer is the target value (we consider a single output). The hidden layers transform the data through weights and biases in such a way as to approximate the functional relationship between inputs and output. The ANN learns the values of these weights and biases that best approximates the relationship during training on labelled data, which is broken down into two stages; feedforward and backpropagation.

Feedforward

Feedforward is the process of connecting the inputs and the output through each layer in the network. For each neuron j in layer l (N_j^l) , compute the following equations:

$$N_j^l = \sum_{i=1}^{l-1} \sigma^l(N_j^{l-1} w_{i,j}^l + b_{i,j}^l)$$

[2] Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S. 2017 [9]
[3] Thomas, R. (2018) [13]
[4] https://jupyter.org/ [21]
[5] http://www.statsmodels.org/stable/ [22]

Where $w_{i,j}^{l}$ and $b_{i,j}^{l}$ represent the weights and biases connecting neuron i in the (l-1)th layer to neuron j in layer l and σ^{l} represents a non-linear activation function. The sum is computed over all neurons in the (l-1)th layer.

This process propagates between each neuron and all the neurons in the previous layer to approximate a function which predicts the target based on the set of features. However, as the parameters of the network (the weights and biases) are randomly initialized, this function will not approximate the relationship with any accuracy. The process of training the ANN to improve its accuracy (reduce the cost or error between predicted values and actual values of the target variable) is done through backpropagation.

Backpropagation

In order to learn the best weights and biases for the model, the ANN must make use of a cost function and an optimization algorithm. The cost function measures how far the original predictions are from the true value, and the optimization procedure changes the parameters in the ANN in order to minimize this cost function. A common cost function for regression (when the target is a continuous variable) is Mean Standard Error (MSE)

$$C = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where y_i is the true output, and $\hat{y}_i$ is the predicted output.

The traditional optimization algorithm used to minimize the cost function is gradient descent, where we try to move down the surface of the cost function to reach a (ideally) global minimum. This process works by calculating the derivative of the cost function with respect to each layer of weights, and updating those weights by the rule:

$$W = W - \alpha\frac{\partial C}{\partial W}$$

Where C= Cost, W = weights, α = learning rate

While we cannot calculate this partial derivative directly, we can use the chain rule to calculate this result based on the partial derivatives:

$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial Output}\frac{\partial Output}{\partial Hidden}\frac{\partial Hidden}{\partial W}$$

In practice however, updating the weights after computing cost on all training samples is too computationally taxing, and a method called stochastic gradient descent (SGD) is used instead. SGD works by updating the weights and biases after only computing the cost from one or a batch of training samples. Whilst normal gradient descent is deterministic and will converge to the same minimum each time it is run, SGD is stochastic since it updates weights after each batch in a shuffled training set. SGD is preferred since it is more computationally efficient and has been shown to converge to a very close approximation of the global minimum in most cases.[6]

[6] L. Bottou and N. Murata. 2002 [14]

For our models we use a variation on SGD called Adam, which separately optimises the learning rate and uses the first and second moments of the parameters to reach an approximate global minimum faster.[7]

The cycle of passing all the data through the network forwards and backwards is called an epoch, and generally ANNs require many epochs before an acceptable level of accuracy is reached.

Theoretical Motivation

Universal Approximation Theorem

The theoretical motivation for the fact that ANNs should be able to perform as accurately on structured data as unstructured data comes from the universal approximation theorem, which states that any neural network with one hidden layer can approximate any continuous function. In 1989 George Cybenko showed that the universal approximation theorem holds when the neural network is constructed using a sigmoid activation function,[8] however this was extended by Kurt Hornik in 1991 who showed that the theorem holds for any activation function, and it is actually the neural network architecture that is responsible for the result. [9]

Theorem: Let $\sigma: \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous activation function. Let $\mathbb{Q}^m$ denote the m dimensional unit hypercube. The space of real valued continuous functions on $\mathbb{Q}^m$ is denoted by $\mathrm{C}(\mathbb{Q}^m)$. Then given any $\varepsilon > 0$ and any function $f \in \mathrm{C}(\mathbb{Q}^m)$ there exists an integer N, bias $b_i \in \mathbb{R}$ and vector of weights $w_i \in \mathbb{R}$ for $i = 1, \ldots, N$ such that we may define:

$$\mathrm{F}(x) = \sum_{i=1}^{N} \sigma(w_i^T x + b_i)$$

Such that:

$$|\, \mathrm{F}(x) - f(x) \,| < \varepsilon$$

That is, any neural network with one hidden layer should be able to approximate any continuous function to a desired accuracy.

Entity Embedding

Applying a neural network architecture to structured datasets involving categorical variables presents a central issue; the function connecting the categorical features to the target may not be continuous, and if that is the case, then the universal approximation theorem does not hold. Traditionally the way to circumvent this problem is a process called one-hot encoding. That is, to transform a single feature of n categories into n distinct features each with a binary value. However, when there is a large number of categories, this one-hot encoded data increases the dimensionality of the data dramatically which can lead to issues of higher computing power. It also treats each categorical variable as independent and thus does not take advantage of any relationships between the categories. For example, the categorical feature "Day of the week" usually has some relationship between week days as opposed to weekend days, which a one hot encoded dataset would not be able to make use of.

[7] Kingma, D.P. and Ba, J., 2014.[15]
[8] Cybenko, G., 1989.. [16]
[9] Hornik, K., 1991. [17]

In order to overcome the problem of continuity without running into computational and dimensionality issues we can use a technique called entity embedding. The process of entity embedding transforms each categorical variable into a dense vector of size D in Euclidian space with continuous valued entries (usually D is much smaller than the number of distinct levels in the categorical feature).

Embedding has gained popularity in machine learning mostly in the context of natural language processing in which words are given a vector representation. Popular word embedding algorithms are Google's Word2vec[10] and Stanford's GloVe,[11] which also act as pretrained corpuses for word co-occurrence and representation. Whilst our datasets do not contain natural language, the underlying principles are the same and have been explored by Guo et al.[12]

Adding an embedding layer to our neural network is a special case of an input layer. We first create a matrix of dimensions $m \times D$ where m is the number of distinct values of the categorical feature c_i, and D is a hyperparameter which specifies the length of the embedding vector. For each possible value of c_i, the embedding vector is recovered as

$$V_i = \sum_j (\delta_{c_i,j} \quad \dots \quad \delta_{c_m,j}) \begin{pmatrix} E_{1,1} & \cdots & E_{1,D} \\ \vdots & \ddots & \vdots \\ E_{m,D} & \cdots & E_{m,D} \end{pmatrix}$$

Where $\delta_{i,j}$ is the Kronecker Delta function defined as

$$\delta_{i,j} = \begin{cases} 0, & if\ i \neq j \\ 1, & if\ i = j \end{cases}$$

With $i, j \in c_1, c_2, \dots, c_m$

In this way the neural network can learn the embedding matrix values $E_{i,d}$ as it would learn regular weight and bias parameters to produce embedding vectors V_i for all possible instances of the categorical feature c_i . The vector of continuous (floating point) values helps with finding relationships between the instances.

Original Categorical Variable	One-Hot Encoded Vector	Entity Embedding Vector
Monday	< 1,0,0,0,0,0,0>	<0.1,0.9,0.9>
Tuesday	< 0,1,0,0,0,0,0>	<0.2,0.9,0.8>
Wednesday	< 0,0,1,0,0,0,0>	<0.3,0.9,0.7>
Thursday	< 0,0,0,1,0,0,0>	<0.4,0.8,0.6>
Friday	< 0,0,0,0,1,0,0>	<0.8,0.9,0.5>
Saturday	< 0,0,0,0,0,1,0>	<0.9,0.1,0.2>
Sunday	< 0,0,0,0,0,0,1>	<0.9,0.2,0.3>

Figure 1: Example of different representations of a categorical variable

[10] Mikolov, Tomas et al. 2013 [11]
[11] Pennington, J et al. 2014 [4]
[12] Guo, C. and Berkhahn, F. 2016 [2]

To build our neural networks with embedding, we simply concatenate the embedding layer, which takes the categorical feature as an input, with a regular neural network that takes the 2 continuous features as an input. The embedding layer becomes the first layer in the network, and a normally constructed network of dense layers follows.

Datasets

We create 3 datasets for the regression task, and 3 datasets for the classification task. We keep these datasets restricted to only 2 or 3 features, with the amount of noise parameterized to keep consistency and provide a good benchmark for neural networks on a simple dataset.

Dataset 1

The first dataset we construct is made of 2 continuous features ranging between 0.0 and 10.0. The relationship between the features and the target follows a polynomial relationship. We include an error term e which is generated from a normal distribution centred at 0 with standard deviation σ which we include as a parameter in order to do experiments with different levels of noise.

$$y = a_1 x_1^2 + a_2 x_2 + e$$

We manually set coefficients a_1 and a_2 to 2 and 5 respectively.

Dataset 2

The second dataset we construct includes the same 2 continuous features, however the relationship is now linear instead of polynomial. We also add a categorical feature ranging between 1 and 7. We choose 7 categories to represent a common categorical feature which is days of the week. Here he relationship between the continuous features and the target follows a linear relationship. We also one-hot encode the categorical feature and manually set coefficients such that weekdays and weekends are encoded equivalently.

$$y = a_1 x_1 + a_2 x_2 + a_{weekday} \sum_{i=1}^{5} c_i + a_{weekend} \sum_{i=6}^{7} c_i + e$$

Where c_i represents the one-hot encoded categorical feature. We again manually set coefficients a_1 and a_2 to 2 and 5 respectively, and we set $a_{weekday} = 1$ and $a_{weekend} = 7$

Dataset 3

The third dataset we construct is exactly the same as dataset 2 however we reintroduce the polynomial relationship in the continuous features.

$$y = a_1 x_1^2 + a_2 x_2 + a_{weekday} \sum_{i=1}^{5} c_i + a_{weekend} \sum_{i=6}^{7} c_i + e$$

Where c_i represents the one-hot encoded categorical feature. We again manually set coefficients a_1 and a_2 to 2 and 5 respectively, and we set $a_{weekday} = 1$ and $a_{weekend} = 7$

Classification

For the classification problems all datasets are constructed in the same way, however we add another layer of conditions which converts the output from a continuous variable to binary variable. We first scale the y values to the range (0,1).

$$y \leftarrow \frac{y - y_{min}}{y_{max} - y_{min}} + y_{min}$$

We then convert the continuous y to a binary variable based on a threshold

$$\mathrm{y} \leftarrow [0 \text{ if } \mathrm{y} < \text{threshold}, \text{else } 1]$$

Changing the value of the threshold allows up to test our models on datasets that have different distributions of the y variable. For the results shown in this report we use a cut-off of 0.5, however for future research we would like to experiment with different distributions of classes.

Models

Statsmodels – dataset 1

As our dataset is constructed through a polynomial combination of features, we choose a baseline model to be polynomial regression which we construct using the python package Statsmodels. Polynomial regression works by finding the optimum values of coefficients $a_{0,1,\ldots,n}$ in the following relationship, which we specify using statsmodels formula API

$$\hat{y}_i = a_1 x_1^2 + a_2 x_2$$

The coefficients are chosen randomly and then updated through the algorithm Ordinary Least Squares (OLS) which attempts to minimize the following loss function

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Since we designed the dataset such that this was the relationship, plus some error term, this polynomial regression model will serve as a good baseline in which to measure the performance of the ANN.

Statsmodels – dataset 2

The basic model for dataset 2 is similar, however we now specify the formula

$$\hat{y}_i = a_1 x_1 + a_2 x_2 + C(x_3)$$

Where the operator C converts the third feature into its one-hot encoded form.

Statsmodels – dataset 3

The basic model for dataset 3 is similar to 2, however we now specify the polynomial term

$$\hat{y_i} = a_1 x_1^2 + a_2 x_2 + C(x_3)$$

Statsmodels - classification

For the classification version of these three problems we instead use the Logit model, which optimises the logistic function

$$\frac{e^t}{1 + e^t}$$

Where t is the linear, or polynomial relationship between variables for each dataset. Once optimised through maximum likelihood estimation, the output for logistic regression is the probability that each observation belongs to the first class. We simply round this number to 0 or 1 in order to convert the continuous output to a binary output which represents its class.

Tensorflow

The table below summarises the hyperparameters used by our neural network models for both regression and classification. For the sake of computation time, for the three datasets in each task we keep all hyperparameters constant except for the size and number of hidden layers, which we roughly optimise through experimentation.

Problem	Hidden layer activation	Output layer activation	Loss function	Epochs	Optimisation algorithm	Number of hidden layers	Size of hidden layers
Regression	Relu	Linear	Mean Squared Error	1000	Adam	[1,2,3,4]	[4,16,32,64,128 ,256,512,1024]
Classification	Relu	Sigmoid	Binary Crossentropy	1000	Adam	[1,2,3,4]	[4,16,32,64,128 ,256,512,1024]

Fig 2: Hyperparameters

The activations function called Rectified Linear Unit activation function (ReLU) has a formulation:

$$a = \max(0, x)$$

The linear output layer activation function simply returns the linear combination of weights and biases, and so essentially means that there is no non-linear activation function in that layer.

The sigmoid activation function is defined as:

$$a = \frac{1}{1 + e^{-x}}$$

Sigmoid is used for classification as it maps the output to a range of between 0 and 1, which we simply round up or down in order to convert the output into a class.

We use MSE which has been defined earlier as the loss function over which to train the neural networks on regression, however we use binary cross-entropy as the loss function for classification which is defined as:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1-y_i) \cdot \log(1-p(y_i)$$

Comparison Algorithm

We use the following algorithm to compare Statsmodels with Tensorflow:

1. Compute cost for Statsmodels and ANN with 1000 observations
2. If ANN and Statsmodels perform equivalently (within 1%) do:
    a. Reduce number of observations by 100 and go back to step 2
    b. Else: go to step 3
3. If ANN performs worse than Statsmodels:
    a. Change number of hidden layers
    b. Change size of hidden layers
    c. Compute cost again and go to step 2
4. If ANN worse than Statsmodels after testing with many variations of number and size of hidden layer:
    a. Cutoff ← Number of Observations

Results

Regression

Dataset 1

Noise	*Statsmodels RMSE*	*Standard deviation of error*	*Neural Network RMSE*	*Standard deviation of error*	*Observations*
0.01	0.01015	0.0004	0.51003	0.0328	1000
0.1	0.09865	0.0035	0.95749	0.0157	1000
1	0.98846	0.0341	1.61335	0.2587	1000
10	10.0319	0.3656	11.2952	0.4177	1000
100	100.181	3.4000	114.773	4.4077	1000

Dataset 2

Noise	*Statsmodels RMSE*	*Standard deviation of error*	*NN + embedding RMSE*	*Standard deviation of error*	*NN no embedding*	*Standard deviation of error*	*Observations*
0.01	0.01009	0.00034	0.28449	0.00549	0.05936	0.02063	1000
0.1	0.09897	0.00338	0.33869	0.00721	0.15977	0.00379	1000

1	1.03705	0.04617	1.10394	0.05223	1.11996	0.03849	1000
10	9.82527	0.33099	10.6233	0.41058	10.9153	0.50527	1000
100	101.918	3.5842	105.785	5.29500	118.275	4.65483	1000

Dataset 3

Noise	*Statsmodels RMSE*	*Standard deviation of error*	*NN + embedding RMSE*	*Standard deviation of error*	*NN no embedding RMSE*	*Standard deviation of error*	*Observations*
0.01	0.01001	0.00042	0.56978	0.01126	0.5035	0.01552	1000
0.1	0.01007	0.00366	0.64496	0.01734	0.6936	0.05338	1000
1	1.00917	0.04054	1.71947	0.07224	1.7983	0.06472	1000
10	10.2089	0.44171	11.9922	0.45304	12.0045	0.43030	1000
100	103.7569	4.14894	109.114	5.31468	120.5246	4.45941	1000

Classification

Dataset 1

Noise	*Statsmodels Accuracy*	*Standard deviation*	*Neural Network Accuracy*	*Standard deviation*	*Observations*
0.01	0.97272..	0.02443	0.97575	0.04211	100
0.1	0.98181..	0.03386	0.97878	0.01786	100
1	0.98181..	0.02674	0.97272	0.04179	100
10	0.97979..	0.01297	0.97878	0.02012	200
100	0.84666	0.02785	0.82666	0.03424	500

Dataset 2

Noise	*Statsmodels Accuracy*	*Standard deviation*	*Neural Network Accuracy*	*Standard deviation*	*NN no embedding Accuracy*	*Standard deviation*	*Observations*
0.01	0.95303..	0.03248	0.95454	0.02364	0.89090	0.03315	200
0.1	0.96212..	0.03084	0.96212	0.03345	0.95454	0.02744	200
1	0.94848..	0.02671	0.95000	0.03435	0.91666	0.03441	200
10	0.76363..	0.04436	0.76060	0.05122	0.67121	0.05023	200
100	0.49393..	0.05785	0.49393	0.07681	0.49848	0.05858	200

Dataset 3

Noise	*Statsmodels Accuracy*	*Standard deviation of error*	*Neural Network Accuracy*	*Standard deviation of error*	*NN no embedding Accuracy*	*Standard deviation of error*	*Observations*
0.01	0.96060..	0.02345	0.97272	0.02859	0.94545	0.02778	200
0.1	0.96818..	0.03782	0.96666	0.03235	0.95454	0.02697	200
1	0.95606..	0.01969	0.95303	0.02190	0.95151	0.03025	200
10	0.94848..	0.02189	0.94545	0.02412	0.92727	0.02975	300
100	0.70707..	0.04012	0.54545	0.03941	0.53030	0.05947	1000

From these results we get an overall picture of how ANN with or without embedding layers compare to traditional statistical methods on small, structured datasets. The main insights we can derive from these results are:

- On small simple structured datasets, ANNs are competitive with logistic regression on classification problems, but are significantly outperformed by linear and polynomial regression on regression problems.
- On datasets constructed with a linear or polynomial relationship, ANNs cannot compete with traditional regression models when there are fewer than 1000 observations.
- On classification datasets with a linear or polynomial relationship, ANNs are able to compete with traditional logistic regression down to as low as 100 observations in some cases.
- ANNs that make use of an embedding layer perform equivalently to those without an embedding layer on datasets with a low level of noise, although there is some evidence to suggest that embedding may improve the model on more complex datatsets.

Regression vs Classification

Our results table has shown that even at 1000 observations, ANNs on each dataset are unable to compete with linear and polynomial regression, however on the classification datasets ANNs perform equivalently even with as little as 100 observations. We believe this may be because classification datasets inherently have a level of non-linearity which regression datasets lack. This gives an advantage to ANNs, as they are able to learn non-linear relationships easily.
It is not in the scope of this research project to investigate this further, but it is nonetheless interesting, and provides a good heuristic for machine learning with artificial neural networks.

Regression

Here we found that on small structured datasets composed of only 2 or 3 features, traditional statistical methods are very hard to beat using an ANN. Even with high levels of noise linear and polynomial regression is able to predict the target of an unseen test set very quickly and accurately. As such we must conclude that when faced with a dataset of this kind, the best approach will not be to use an ANN. This makes sense as linear models are hard to beat when the true relationship between the dependent and independent variables are linear.

Classification

Here we have found that the lower bound in number of observations for ANNs to perform competitively with a traditional benchmark is as low as 100 in many cases. This is an interesting result that we did not expect to find considering ANNs have mostly been used on much larger datasets. This result could indicate to industry that for machine learning pipelines in classification, an ANN model may still be useful for the occasional dataset with few observations.

Embedding

Motivated by recent machine learning blogs and Kaggle competitions which have championed the use of entity embeddings for categorical data, we wanted to explore whether or not an embedding layer in an ANN would improve a model on a small dataset. Our results are mostly inconclusive for these simple datasets, however we believe that in more complex datasets entity embedding would significantly improve the model. Evidence for this comes from the fact that when we increased the noise on our datasets the ANN with an embedding layer became significantly better. We also

performed some pilot experiments on a more complex dataset from the Python machine learning library Scikit-learn and also found that a model with embedding performed better. For future study we recommend increasing the complexity of the datasets in order to determine whether or not entity embedding can improve a models performance.

Future Research

As this project was restricted to only 6 weeks we were not able to perform as in-depth an analysis as we had hoped. Throughout the beginning of the project I had to familiarise myself with the Keras/Tensorflow machine learning framework, and there were many troubleshooting issues related to embedding layers which could only be resolved by downgrading the Keras library. Time constraints also meant that we restricted ourselves to only 3 datasets in each task with the target feature constructed through a simple linear or polynomial relationship with the other features. For future research we hope that more complex datasets can be explored, specifically with the concept of entity embeddings as we believe this technique has the potential to significantly improve ANNs on structured data. Our results showed a glimpse of this since our results on the regression dataset with high stochastic noise showed that a model with embeddings was significantly better than a model without. With more complex datasets we believe this result can be shown more clearly.

Code

The python code for this project can be found at: https://github.com/DanC777/AMSI-VRS-Project

References

[1] Chollet, F. (2017). *Deep learning with Python*. 1st ed. Greenwich, CT, USA: Manning Publications Co.

[2] Guo, C. and Berkhahn, F., 2016. Entity embeddings of categorical variables. *arXiv preprint arXiv:1604.06737*.

[3] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S. and Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (pp. 3111-3119).

[4] Pennington, J., Socher, R. and Manning, C. (2014). *GloVe: Global Vectors for Word Representation* Nlp.stanford.edu. Available at: https://nlp.stanford.edu/projects/glove

[5] Tamang, A. (2017). *Learning Entity Embeddings in one breath – Apil Tamang – Medium*. Medium. Available at: https://medium.com/@apiltamang/learning-entity-embeddings-in-one-breath-b35da807b596

[6] Renom.jp. (2018). *Application of Entity Embedding Layer*. Available at: https://www.renom.jp/notebooks/tutorial/embedding/entity_embedding/notebook.html

[7] Pasini, A., 2015. Artificial neural networks for small dataset analysis. *Journal of thoracic disease*, *7*(5), p.953.

[8] Dreiseitl, S. and Ohno-Machado, L., 2002. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, *35*(5-6), pp.352-359.

[9] Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S., 2017. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems* (pp. 971-980).

[10] Saad, D., 1998. Online algorithms and stochastic approximations. *Online Learning*, *5*.

[11] Mikolov, T., Chen, K., Corrado, G. and Dean, J., 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[12] Goldberg, Y. and Levy, O., 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

[13] Thomas, R. (2018). *An Introduction to Deep Learning for Tabular Data · fast.ai*. Fast.ai. Available at: https://www.fast.ai/2018/04/29/categorical-embeddings

[14] L. Bottou and N. Murata. Stochastic approximations and efficient learning. (2002) *The Handbook of Brain Theory and Neural Networks*, 2nd ed. The MIT Press, Cambridge, MA, 2002.

[15] Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[16] Cybenko, G., 1989. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*, pp.183-192.

[17] Hornik, K., 1991. Approximation capabilities of multilayer feedforward networks. *Neural networks*, *4*(2), pp.251-257.

[18] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

[19] TensorFlow. (2019). *TensorFlow*. Available at: https://www.tensorflow.org/

[20] Colab.research.google.com. (2019). *Google Colaboratory*. Available at: https://colab.research.google.com

[21] Project Jupyter. (2019). Available at: https://jupyter.org/

[22] Perktold, J., Seabold, S. and Taylor, J. (2017). *StatsModels: Statistics in Python — statsmodels 0.9.0 documentation*. Statsmodels.org. Available at: http://www.statsmodels.org/stable/