FINAL TERM PAPER

Topics in Econometrics and Statistics, Summer, 2022

By Tim Mensinger

University of Bonn

Matriculation No.: 2916323

Online repository

https://github.com/timmens/neural-net

In this paper, I investigate the predictive performance of neural networks. I compare unregularized and regularized networks, where in the latter case, I place an ℓ_1-type penalty on the weights. For the unregularized case, I discuss convergence rates. Using a simulation study, I investigate whether neural networks can recover sparse structures in high-dimensional settings and if their performance is superior to other methods. I find that for nonlinear DGPs, neural networks outperform linear models but are beaten by boosting, whereas, for linear DGPs, OLS and the Lasso beat neural networks and boosting. Computer codes and additional materials can be found in the online repository: https://github.com/timmens/neural-net.

CONTENTS

1. Introduction. With the rise of cheap computing, research and application of computer-aided statistical algorithms have skyrocketed. In particular, machine learning, a sub-area of computational statistics that focuses on using algorithms, has experienced the most significant increase in research and its direct application in the industry. Given this success, other research areas are trying to figure out how to incorporate these successful methods to improve their research; see, for example, Varian (2014) and Athey and Imbens (2019). This task turns out harder than expected since research in economics –or other disciplines– is (usually) interested in inference, but machine learning is (usually) used for prediction. Some authors combined problems in econometrics, particularly causal inference, with machine learning methods. Most notable is the paper series by Susan Athey, Stefan Wager, and Guido Imbens that solves the estimation of heterogeneous treatment effects using tree-based machine learning methods; see Athey and Imbens (2016) and Wager and Athey (2018). A different approach is taken in the literature on doubly robust estimation of structural parameters; see for example Chernozhukov et al. (2018). The estimators studied there require the estimation of nuisance functions under high-level assumptions on the convergence rates. Since these nuisance functions are often conditional expectations, this opens up the possibility of using arbitrary machine learning methods for the estimation. A question that remains to be answered is whether machine learning methods perform better on typical economic data sets.

The rest of this paper is structured as follows: In the remaining part of the introduction, I will provide a primer on neural networks. Section 2 lists fitting procedures and regularization strategies for neural networks. These parts can be skipped if the reader is familiar with neural networks. In section 3, I consider approximation theorems and convergence rates. Section 4 presents a simulation study comparing neural networks with linear models in different scenarios. Section 5 concludes.

1.1. *A Primer on Neural Networks.* In the remaining parts of this section, I provide a small primer on feed-forward neural networks. I discuss the connection between their graphical and mathematical representation and common concepts such as the activation function.

Fitting techniques are presented in the next section. For a detailed description of neural networks, see Goodfellow and Courville (2016); Murphy (2012) or Hastie, Tibshirani and Friedman (2008).

On a high-level, neural networks can be thought of as parametrized functions $f(\cdot;\theta) : \mathbb{R}^p \to \mathbb{R}^l$ mapping some feature vector $x \in \mathbb{R}^p$ to the outcome space, with (potentially) high-dimensional parameter vector $\theta \in \Theta$, that ship with efficient procedures to calibrate this parameter on data.

The approximation quality of neural networks has been studied extensively in the literature. Several universal approximation theorems have been proven, which show that under certain conditions on the network architecture, they can approximate any continuous function arbitrarily well. This will be made more precise in Section 3.

General Notion. In the remainder of this section, let us assume that the parameter vector θ has already been estimated. How does a neural network generate predictions?

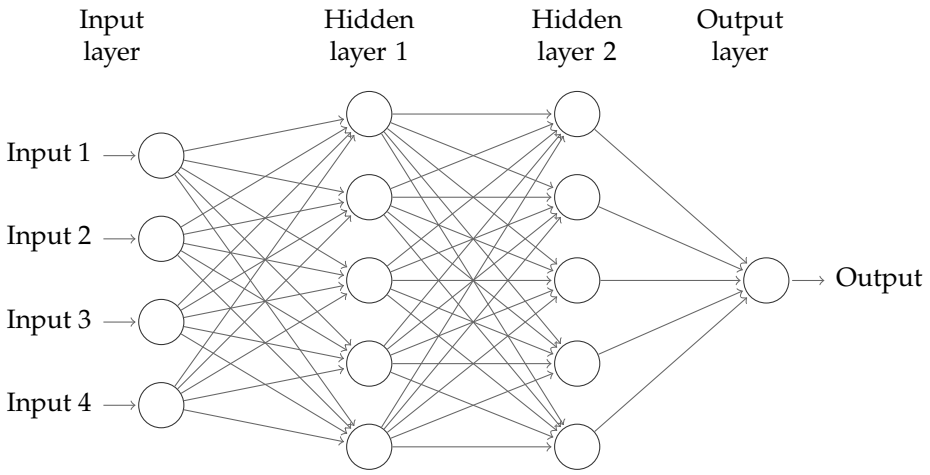

Fig 1. Depiction of a neural network. The network has 4 inputs, two hidden layers with 5 nodes each and a single output node.

Consider the depiction of a two-layer network in figure 1. This neural network represents a function $f : \mathbb{R}^4 \to \mathbb{R}$. The input layer represents the feature vector; similarly, the output layer represents the outcome variable. The network is called a two-layer network because there are two hidden layers. The number of nodes in each

layer is called the depth of that layer, and the number of hidden layers is called the depth of the network. In this paper, I consider fully-connected networks, which means that each node in a previous layer is connected with each node of the next layer. To build the bridge to the mathematical formulation, let us focus on the connection between the input layer and a single node in the first hidden layer. This is visualized in Figure 2.

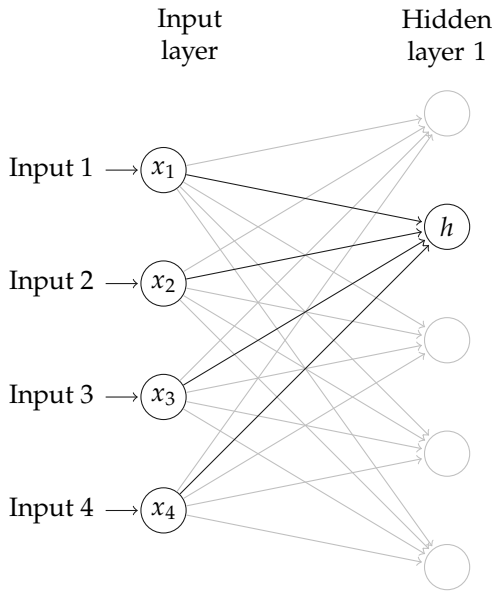

Fig 2. Depiction of the connection between the input layer and a single hidden node.

Consider an arbitrary node h from the first hidden layer. The value that h attains after the network receives an input $x = (x_1, x_2, x_3, x_4)^\top$, is defined as

$$h(x) = \sigma(w^\top x + b),$$

where $w \in \mathbb{R}^4$ denotes the weights of the edges and $b \in \mathbb{R}$ the bias term, which is associated with each hidden node. σ denotes the activation function. Activation functions are motivated by biological processes in the brain. Human neurons require their input signal to exceed some threshold before they send out new signals; see Goodfellow and Courville (2016). Activation functions try to model this

behavior by outputting a low value when $w^\top x$ is smaller than the bias b, and outputting a larger value when $w^\top x$ overcomes the bias. Traditionally the most common activation function is the Sigmoid function $\sigma(z) = 1/(1 + \exp(-z))$ which is depicted in Figure 3.

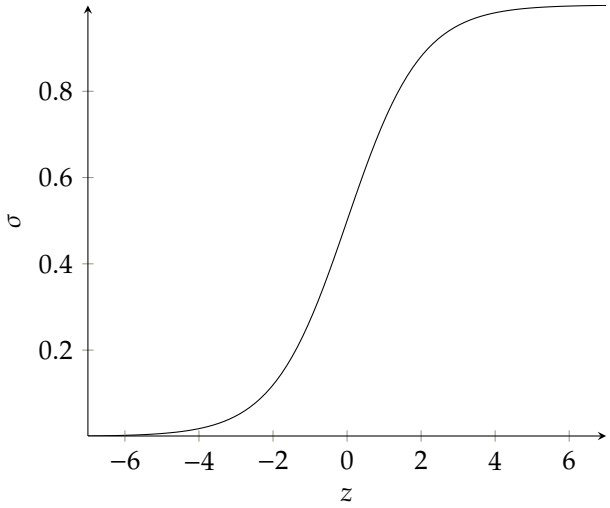

FIG 3. Sigmoid function.

To compute the value of a node in the second hidden layer, we need more notation. Consider Figure 4, and let me now denote an arbitrary node of the second hidden layer by h. If we had already computed the values of the nodes $\mathbf{h}_1 = (h_1, \ldots, h_5)^\top$ the same logic would give the value of $h$as before: $h = \sigma(w_h^\top x + b_h)$. Here I represent the bias term for node h as b_h and the weights connecting the first layer with node h as w_h. Letting w_{ij} denote the weights connecting all nodes in layer i with node j of layer $i + 1$, we know from before that

$$\mathbf{h}_1(x) := \begin{bmatrix} \sigma(w_{11}^\top x + b_{11}) \\ \vdots \\ \sigma(w_{15}^\top x + b_{15}) \end{bmatrix}.$$

We can simplify this by defining a weight matrix for each layer. For the first hidden layer, we have $W_1 := (w_{11} \ldots, w_{15})^\top$. Allowing element-wise operations on σ we get $\mathbf{h}_1(x) := \sigma(W_1 x + b_1)$ and ultimately for

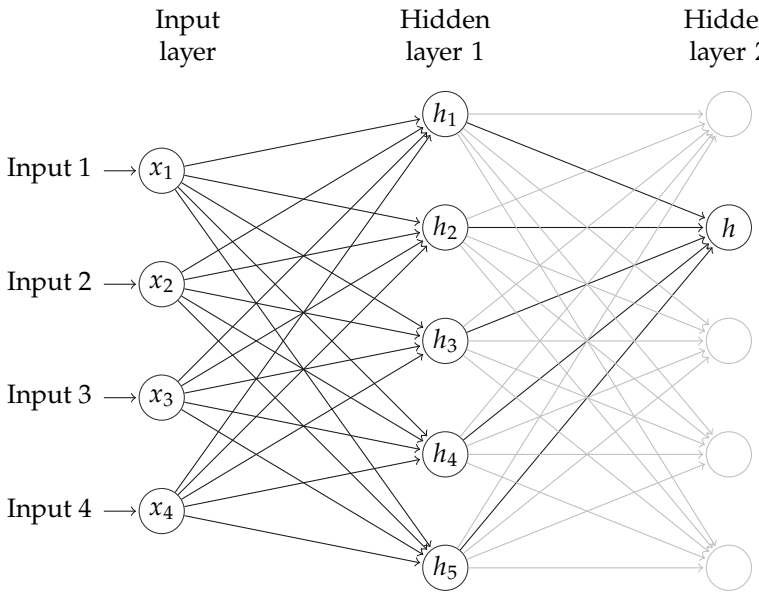

Fig 4. Depiction of the connection between the input layer and a single node in the second hidden layer.

the value of h we have $h(x) = \sigma(w_h^\top \mathbf{h}_1(x) + b_h)$. Combining all of the above, we can write down the analytical representation of this neural network. Define W_2 analogously to W_1 and let a_0 and a denote the bias term and weight vector for the last layer, respectively. Then,

$$\begin{aligned} f(x) &= a_0 + a^\top \mathbf{h}_2(x) \\ &= a_0 + a^\top \sigma(W_2\, \mathbf{h}_1(x) + b_2) \\ &= a_0 + a^\top \sigma\big(W_2\, \sigma(W_1\, x + b_1) + b_2\big). \end{aligned}$$

Extensions. The network I presented above is the bare-bone version of neural networks. A popular model component replaced by modern versions is the activation function; a comparison is presented in Hara and Nakayamma (1994). The optimal choice of the activation function depends on the problem at hand, but the state-of-the-art default is the ReLU function, depicted in Figure 5. Krizhevsky, Sutskever and Hinton (2017) provide empirical evidence that the ReLU function can improve upon the classical Sigmoid function by accelerating the convergence of the fitting procedure.

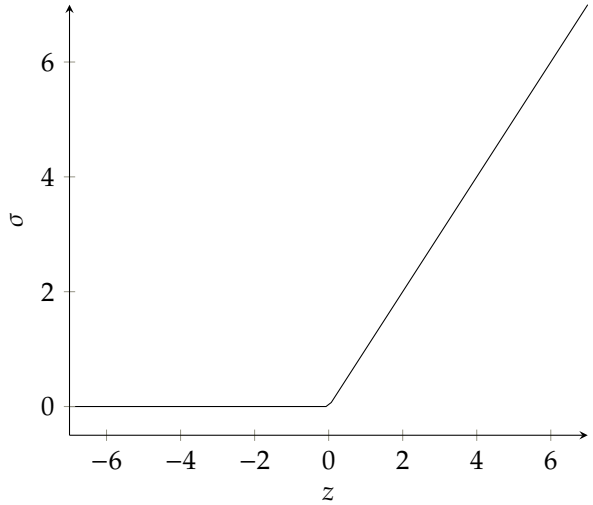

Fig 5. Rectified linear unit (ReLU) activation function.

2. Fitting. In this section, I discuss general fitting techniques focusing on estimators that induce a non-convex but differentiable loss function. I explain approaches used when training neural networks and how they are extended to regularize the fitting process. I assume that an IID dataset $\{(x_i, y_i)|i = 1, \ldots, n\}$ is available with features $x_i \in \mathbb{R}^p$ and conditional expectation $f(x) = \mathbb{E}[y_i|x_i = x]$. If not otherwise stated, I assume that all necessary moments exist. The problem I consider in this paper is to find a good approximation, in a mean-squared-error sense, of the conditional expectation f using the data. In Section 1, I argued that this problem arises naturally in econometrics when estimating nuisance functions.

2.1. *General Fitting Techniques.* Let us consider a generic class of parametrized approximation functions $\{g|g(\cdot;\theta,\eta) : \mathbb{R}^p \to \mathbb{R}, \theta \in \Theta(\eta)\}$, with θ denoting a given parametrization and η a hyper-parameter configuration. Hyper-parameters denote model design choices that are not estimated at the same time as the parameters. For neural networks, a typical hyper-parameter is the network architecture and the class of activation functions. Similarly, for boosting, the class of weak-learners and the learning rate are viewed as fixed during the estimation. Grid-based methods like cross-validation and similar traditional approaches can be used to estimate such hyper-

parameters on the data; however, these methods quickly become infeasible if there are more than two variables to calibrate. This is because the number of grid points grows exponentially with the dimensions, making the problem computationally intractable. Modern approaches utilize Bayesian optimization techniques to deal with high-dimensional hyper-parameter spaces; see for example Snoek, Larochelle and Adams (2012). In the following, I will assume a fixed hyper-parameter configuration, and for the sake of clarity, I will thus suppress any dependence on η.

Loss function. What expression do we need to minimize to get an estimator of θ? From statistical decision theory, we know that the conditional expectation f solves the problem

$$\min_h \mathbb{E}\left[\ell\left(y_i, h(x_i)\right)\right],$$

when $\ell(y, \hat{y}) = (y - \hat{y})^2$ is the squared error loss function. Given our class of approximation functions, we can parametrize this problem via θ

$$\min_{\theta \in \Theta} \mathbb{E}\left[\ell\left(y_i, g(x_i; \theta)\right)\right].$$

Because we do not know the joint distribution of (x_i, y_i), this object is intractable; however, we can use a sample-analogue or non-linear least-squares approach to define the estimator as the solution to a minimization problem containing only observed quantities

$$\text{(MP)} \qquad \min_{\theta \in \Theta} \sum_{i=1}^{n} \ell\left(y_i, g(x_i; \theta)\right).$$

This defines the empirical loss function $L(\theta) = \sum_{i=1}^{n} \ell\left(y_i, g(x_i; \theta)\right)$ that we wish to minimize. Define $\hat{\theta} := \operatorname{argmin} L(\theta)$ as the parameter that solves MP, and $\hat{f}(x) := g(x; \hat{\theta})$ as the estimator of f. Notice that we can make two main mistakes during the estimation. First, because we work on a finite sample, $\hat{\theta}$ will not minimize the expected loss, and second, f may not be representable using the class of approximation functions. In Section 3, I discuss results on the representation properties of neural networks, indicating that the latter problem can be neglected for large enough models.

Minimization. How do we minimize $L(\theta)$? An optimal strategy depends on the class of approximation functions. In the linear case $g(x;\theta) = x^\top\theta$, we know that L is convex and that its minimizer corresponds to the ordinary least-squares estimator. In the general case g, may induce L to be non-convex. Thus, there may not be an analytical solution, and there may be multiple local minima. In these settings, one commonly uses iterative procedures of the form as specified in Algorithm 1. We choose an initial starting value of the parameter

Algorithm 1 Iterative optimization procedure

1: $\theta_0 \leftarrow$ initial value
2: **while** not converged **do**
3: $\theta_{k+1} \leftarrow O(\theta_k, L)$
4: **end while**

and then iteratively apply an updating procedure O to the parameter. If we assume more structure on L we can use procedures with faster convergence rates. A common assumption is differentiability, wherein the procedure uses gradient information during the updating step. The most known case is the gradient descent algorithm that performs the following updating rule

$$O(\theta_k, L) = \theta_k - \eta_k \nabla L(\theta_k),$$

where $\eta_k > 0$ governs the learning speed. Many modern optimization procedures derive from the gradient descent algorithm, such as stochastic gradient descent or Adam; see e.g., Kingma and Ba (2014). These optimizers are well studied, and convergence proofs exist under many different sets of assumptions on L and $\{\eta_k\}$. For example, convergence to a local minimum is achieved if L is twice continuously differentiable with Lipschitz gradient; see Lee et al. (2016).

Contrary to intuition, using second-order information does not necessarily improve the procedure. For large models, the Hessian matrix of L can be very expensive to compute, as closed-form expressions are rarely available. In this case, the gain in accuracy has to offset the loss in computational time. This is why optimizers using second-order information are rarely used when training models with many parameters, compared to small-scale models, like logistic regression, which is commonly solved using the second-order Newton-Raphson algorithm; see e.g., Pedregosa et al. (2011).

2.2. *Fitting of Neural Networks.* The complexity of training a neural network can be attributed to two main problems. One, the (computationally efficient) calculation of the gradient $\nabla L(\theta)$, and two, the avoidance of overfitting.

Backpropagation. As we have seen above, computation of the gradient $\nabla L(\theta)$ is essential to training a neural network. Since neural networks are constructed using a composition of nonlinear activation functions applied to matrix-vector products, the structure of the derivative can be derived using the chain-rule. Unfortunately, naive implementations of this procedure are so inefficient that they cannot be used to train large neural networks. The term backpropagation was coined by Rumelhart, Hinton and Williams (1986), who proposed a computationally efficient algorithm for calculating the gradient. For a detailed explanation of the backpropagation technique and its connection to automatic differentiation, see Section 6.5 of Goodfellow and Courville (2016).

Regularization. The number of parameters in a neural network can explode quickly. The network depicted in figure 1 contains $4 \times 5 + 5 \times 5 + 5 \times 1 = 50$ parameters (excluding the biases). In the case of a 100-dimensional feature vector and two hidden layers with 100 hidden nodes each, the number of parameters would grow to $100 \times 100 + 100 \times 100 + 100 = 20100$. Clearly, overfitting is an acute hazard in large networks. Many methods have been developed to mitigate this problem, such as stochastic gradient descent (Ruppert (1985)), dropout regularization (Srivastava et al. (2014)), early stopping (Caruana, Lawrence and Giles (2000)) or weight decay (Krogh and Hertz (1991)).

In my simulations, I use two different approaches to fit neural networks. In the first, which I call *unregularized*, I use the Adam optimizer which works similar as stochastic gradient descent. In the second, which I call *regularized*, I use the same optimizer and additionally place an ℓ_1-penalty on the weights connecting the input layer and the first hidden layer. This approach is similar to weight-decay, which uses an ℓ_2-penalty on all weights.

3. Rates. In this section, I consider two different kinds of results. First, I present representation results that answer the question of

which classes of functions can be approximated by a neural network. Second, I consider convergence results that answer the question if a neural network can learn a function from data.

3.1. *Approximation Theorems.* When we use neural networks to estimate functional relationships, we hope that the network can approximate the true function as closely as possible. A minimal assumption that can give us hope would be if we could be assured that there exists a neural network that can approximate the target function with some degree of accuracy. The well known Weierstraß theorem provides this result for any continuous real-valued function f defined on a closed domain $[a, b]$: For every $\epsilon > 0$ there exists a polynomial q_ϵ such that $\sup_x |f(x) - q_\epsilon(x)| < \epsilon$; for a proof consult any advanced real analysis textbook. Neural networks are more flexible than polynomials in their architecture; one can alter the number of hidden neurons per layer, the number of layers, or the class of activation functions. Here I present two different results. In the first, we consider a class of networks with one hidden layer that can grow arbitrarily large. In the second, I consider a class of networks with fixed width per layer that can grow arbitrarily deep. In both cases, the result is that under regularity conditions, networks can approximate any continuous functions arbitrarily well.

THEOREM 3.1. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-polynomial continuous function. Let $\mathbb{N}_p^\sigma$ represent the class of neural networks with actiation function σ, input dimension p and one hidden layer with an arbitrary number of neurons. Then $\mathbb{N}_p^\sigma$ is dense in the space of continuous functions $[0, 1]^p \to \mathbb{R}$, with respect to the uniform norm.*

PROOF. See Cybenko (1989); Hornik (1991); Pinkus (1999). □

To get more intuition on why this result holds, consider a one-layer network with h hidden neurons

$$f(x) = a_0 + \sum_{l=1}^{h} a_l \cdot \sigma(w_l^\top x + b_l).$$

By cleverly setting the activation function and parameters, we can reconstruct basis functions. Letting h tend to infinity, the asymptotic

approximation quality should therefore be the same as of other series approximators, e.g., the Fourier series.

THEOREM 3.2. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-affine continuous function such that there exists some $x \in \mathbb{R}$ for which σ is continuously differentiable at x with $\sigma'(x) \neq 0$. Let $\mathbb{N}^{\sigma}_{p,h}$ denote the class of networks with activation functions σ, input dimension p and an arbitrary number of hidden layers that each have h number of neurons. Then $\mathbb{N}^{\sigma}_{p,h}$ is dense in the space of continuous functions $[0,1]^p \to \mathbb{R}$ if $h \geq p+3$, with respect to the uniform norm.*

PROOF. See Kidger and Lyons (2020). □

Similar to the Weierstraß theorem and polynomials, these theorems motivate the usage of neural networks for approximation tasks. However, existence results do not help when choosing the network architecture, and more importantly, they do not show that a neural network can learn a structure from data.

3.2. *Convergence Rates.* In a strict sense, neural networks are parametric estimators with (potentially) many parameters. Assuming that the width or depth of a network can grow with the sample size, these estimators can be analyzed in a nonparametric setting. In nonparametric analysis, it is common to assume that the target function is k-smooth. In this setting, the optimal minimax rate for the L_2 prediction error is $O(n^{-2k/(2k+d)})$; see Stone (1982). The applications where neural networks had the most significant impact require very large input dimensions d, e.g., image classification, where a feature vector can consist of 512×512 pixels. In these scenarios even huge sample sizes are not sufficient to compensate for the slow rates. Thus, a natural question is whether these rates persist for neural networks and if they can be approved when imposing more structure on the target function.

While these theoretical considerations would be by far the most exciting part, discussing them in detail would go beyond the scope of this paper. Nevertheless, let me present some recent papers tackling these questions.

Bauer and Kohler (2019) show that if the target function satisfies a (p, C)-smooth generalized hierarchical interaction model of given

order d^*, for $d^* \leq d$, a deep neural network estimator defined via the solution to the least-squares problem MP can circumvent the curse-of-dimensionality and generate rates of the form

$$O\left(n^{-\frac{2p}{2p+d^*}}\right),$$

which can be much faster than the standard rate if $d^* \ll d$.

In practice, neural networks are not estimated by directly solving the minimization problem MP, but through (derivations of) the gradient descent algorithm; see Section 2. Bauer and Kohler (2019) do not account for this different estimation scheme. Braun, Kohler and Walk (2019) consider single-layer neural network estimators where the weights are learned using a gradient descent procedure. Assuming that the target function follows a projection pursuit model with (p, C)-smooth base functions, they show that a neural network fitted via gradient descent achieves an L_2 prediction error rate

$$O\left(\left[\frac{\log^3 n}{n}\right]^{\frac{2p}{2p+1}}\right).$$

Notably, this rate does not depend on the input dimension d. However, this is not a surprising result since the projection pursuit model's structure is very similar to the structure of a one-layer neural network.

4. Simulation. In this section, I discuss the simulation part of this paper. First, I discuss the study design, the employed fitting methods, and its results. Then I talk about the difficulties of implementing industry-standard machine learning methods. All computer codes needed to reproduce the simulation are available in the online repository `http://github.com/timmens/neural-net`.

4.1. *Study Design.* The simulation is run over three different data generating process types and three levels of the sample size $n \in \{100, 1.000, 10.000\}$. For each specification, I simulate 100 iterations. In each iteration, a model is fitted using training data of size n and its predictions are evaluated on a large testing set that is simulated once

for each specification. I measure the average square error of the predictions on the testing set. The following data generating processes are implemented.

Linear. For the linear model I simulate IID data with features $x_i \sim \mathcal{U}[-1,1]^p$, where $p = 30$ for all sample sizes.[1] The outcomes are computed using $y_i = \beta^\top x_i + e_i$, where $e_i \sim \mathcal{N}(0, 1/4)$ and β is drawn randomly once such that its ℓ_2-norm is 1.

Linear High-dimensional. For the high-dimensional case I simulate IID data with features $x_i \sim \mathcal{U}[-1,1]^p$, where $p = \lfloor n/10 \rfloor$. To incorporate sparsity I randomly select 10% of the features to be informative. Let $x_{i\mathcal{I}}$ denote the vector containing the informative features. Then the outcomes are computed using $y_i = \beta^\top x_{i\mathcal{I}} + e_i$, where $e_i \sim \mathcal{N}(0, 1/4)$.

Nonlinear High-dimensional. For the nonlinear high-dimensional case I simulate IID data with features $x_i \sim \mathcal{U}[-1,1]^p$, where $p = \lfloor n/10 \rfloor$. To incorporate sparsity I select 10% of the features to be informative. Let $x_{i\mathcal{I}}$ denote the vector containing the informative features. Then the outcomes are computed using $y_i = \prod_{j \in \mathcal{I}} x_{ij} + e_i$, where $e_i \sim \mathcal{N}(0, 1/4)$ and β is drawn randomly once for each parameter dimension such that its ℓ_2-norm is 1.

4.2. *Methods.* I employ four different fitting methods in this study.

[1] By $\mathcal{U}[-1,1]^p$ I denote the p-dimensional multivariate uniform distribution on $[-1,1]^p$ with independent components.

OLS	An ordinary least squares model with intercept.
Lasso	A Lasso model with intercept that uses 3-fold cross-validation to determine the penalty parameter over the grid $(0.1, 0.001, 0.0001)$.
Net	A neural network with ReLU activation functions and two hidden layers, where the first layer has $p/2$ nodes and the second $p/4$. The network is trained over 100 epochs using the Adam optimizer.
RegNet	A neural network with ReLU activation functions and five hidden layers with $\max(2, \lfloor s \cdot p \rfloor)$ nodes, where $s \in (0, 1)$ is a sparsity level set to 0.01. Further, the weights connecting the input layer and first hidden layer are penalized using an ℓ_1-norm. The penalty parameter is set to 0.05. The network is trained over 100 epochs using the Adam optimizer.
Boosting	A boosting algorithm with tree weak-learners of depth 2. The algorithm is run over 1.000 iterations. In each split in the regression tree a randomly selected subset of 20% of the features is considered.

REMARK. *Given the sparse nature of the high-dimensional data generating processes, one could argue that incorporating this information into the regularized network via a sparsity level parameter is unfair. This is a design choice I had to take since otherwise, the training would take too long; note that for* $n = 10.000$ *we have* $p = 1.000$ *features, which combined with two hidden layers of size* 500 *and* 250 *would yield* 625.000 *parameters, which cannot be trained in a simulation context on a standard computer.*

4.3. *Results.*

Implementation Remarks.

Training Time. When implementing simulation exercises that use many samples and features, the fitting machine learning algorithms can result in long running times. The training time of state-of-the-art models can take weeks on special hardware. Running these algorithms on Laptop CPUs can increase the training time, so that researchers cannot reproduce industry-standard results since the training time becomes infeasible. In my simulation, I had to choose relatively small networks to circumvent this problem. Even then, on my

2-core machine, the simulation ran for 8 hours. A simple solution is to run the code on GPUs, which can produce speed-ups of up to 20.

Theory and Practice. There is a further problem that hinders the soundness of classical simulation studies. Modern machine learning algorithms are implemented modularly, and each component is optimized to achieve good results even under default arguments, which sometimes set hyper-parameters dependent on the data. Further, the actual implementation and application of these algorithms usually deviate from the mathematical definition. This loosens the connection between Monte Carlo results and theoretical arguments since the researcher may not even be aware that a method is optimizing some hyper-parameter by default. One could argue that this calls for libraries that only implement the theoretical specification to create a more concrete connection between simulation results and theoretical arguments; however, this would miss the point that in practice, people use algorithms with complex defaults. This is a prime example where the gap between theory and practice complicates a complete scientific analysis.

Custom Implementation. Unrelated to the simulation study, I implemented a general neural network and its fitting procedure to understand the technical details better. It is designed for classification and regression and tested on the MNIST data set containing hand-written digits. More details on this mini-project are described in the online repository.

5. Conclusion. In this paper, I tried to give an overview of neural networks and their fitting procedures in the context of econometrics. The simulation study provided evidence that neural networks can outperform linear models in specific scenarios but also showed that other machine learning algorithms can easily beat them. Considering under which circumstances neural networks are successful in the industry, it seems that they require a large number of training samples and architectural fine-tuning by machine learning engineers. In this light, neural networks are not algorithms that one passes a data set to and expects good results. Achievement of good results requires the correct setting of hyper-parameters and optimizers. The theory on the convergence of neural networks has just started considering some of

these options; e.g., Braun, Kohler and Walk (2019). However, the gap between what is done in practice and what is analyzed theoretically remains large.

References.

Athey, S. and Imbens, G. (2016). Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences* **113** 7353-7360.

Athey, S. and Imbens, G. W. (2019). Machine Learning Methods That Economists Should Know About. *Annual Review of Economics* **11** 685-725.

Bauer, B. and Kohler, M. (2019). On deep learning as a remedy for the curse of dimensionality in nonparametric regression. *The Annals of Statistics* **47** 2261 – 2285.

Braun, A., Kohler, M. and Walk, H. (2019). On the rate of convergence of a neural network regression estimate learned by gradient descent. *arXiv: Statistics Theory*.

Caruana, R., Lawrence, S. and Giles, C. (2000). Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *Advances in Neural Information Processing Systems* (T. Leen, T. Dietterich and V. Tresp, eds.) **13**. MIT Press.

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W. and Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal* **21** C1-C68.

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

Goodfellow, Y. Ian; Bengio and Courville, A. (2016). *Deep learning*. MIT Press, Cambridge, MA, USA.

Hara, K. and Nakayamma, K. (1994). Comparison of activation functions in multilayer neural network for pattern classification. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)* **5** 2997-3002 vol.5.

Hastie, T., Tibshirani, R. and Friedman, J. (2008). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4** 251-257.

Kidger, P. and Lyons, T. (2020). Universal Approximation with Deep Narrow Networks. In *Proceedings of Thirty Third Conference on Learning Theory* (J. Abernethy and S. Agarwal, eds.). *Proceedings of Machine Learning Research* **125** 2306–2327. PMLR.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2017). ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **60** 84–90.

Krogh, A. and Hertz, J. (1991). A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems* (J. Moody, S. Hanson and R. P. Lippmann, eds.) **4**. Morgan-Kaufmann.

Lee, J. D., Simchowitz, M., Jordan, M. I. and Recht, B. (2016). Gradient Descent Converges to Minimizers.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT press, Cambridge, MA.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **12** 2825–2830.

Pinkus, A. (1999). Approximation theory of the MLP model in neural networks. *Acta Numerica* **8** 143 - 195.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature* **323** 533-536.

Ruppert, D. (1985). A Newton-Raphson Version of the Multivariate Robbins-Monro Procedure. *The Annals of Statistics* **13**.

Snoek, J., Larochelle, H. and Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems* (F. Pereira, C. J. Burges, L. Bottou and K. Q. Weinberger, eds.) **25**. Curran Associates, Inc.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* **15** 1929–1958.

Stone, C. J. (1982). Optimal Global Rates of Convergence for Nonparametric Regression. *The Annals of Statistics* **10** 1040 – 1053.

Varian, H. R. (2014). Big Data: New Tricks for Econometrics. *Journal of Economic Perspectives* **28** 3-28.

Wager, S. and Athey, S. (2018). Estimation and Inference of Heterogeneous Treatment Effects using Random Forests. *Journal of the American Statistical Association* **113** 1228-1242.