Applied Causal Inference Powered by ML and AI

Victor Chernozhukov* Christian Hansen† Nathan Kallus‡
Martin Spindler§ Vasilis Syrgkanis¶

March 5, 2025

Publisher: Online
Version 0.1.1

* MIT
† Chicago Booth
‡ Cornell University
§ Hamburg University
¶ Stanford University

Predictive Inference via Modern Nonlinear Regression

8

"Nowhere is it written on a stone tablet what kind of model should be used to solve problems involving data."

– Leo Breiman [1].

Here we discuss nonlinear regression methods based on tree models and (deep) neural network models. Tree-based methods include regression trees, random forests, and boosted trees. Regression trees are great for exploration and explainable analytics, while random forests and boosted trees are great predictive tools for structured data and data sets of intermediate size (say, up to several million observations). Neural networks are extremely flexible nonlinear regression methods and are particularly successful for data sets of larger size.

8.1 Introduction

We are interested in predicting an outcome Y using raw regressors Z, which are k-dimensional. The best prediction rule $g(Z)$ under square loss is the conditional expectation function (CEF) of Y given Z:

$$g(Z) = \mathrm{E}(Y \mid Z).$$

In previous chapters, we used best linear prediction rules to approximate $g(Z)$ and linear regression or Lasso regression for estimation. Now we consider nonlinear prediction rules to approximate $g(Z)$, focusing on tree-based methods and neural networks.

The use of best prediction rules (CEFs) is not just important for generating good predictions but is crucial for causal inference. Identification of causal parameters such as ATEs via conditioning strategies requires us to work with CEFs rather than with best linear prediction rules. Previously we tried to make best linear prediction rules flexible to try to approximate best prediction rules. Here we explore fully nonlinear strategies.

8.2 Regression Trees and Random Forests

Introduction to Regression Trees

Regression trees are based on partitioning the regressor space (the space where Z takes on values) into a set of rectangles. A simple model is then fit within each rectangle.

The most common approach fits a simple constant model within each rectangle, which corresponds to approximating the unknown function by a "step function." Given a partition into M regions, denoted $R_1, \ldots, R_M$ the approximating function when a constant is fit within each rectangle is given by

$$f(z) = \sum_{m=1}^{M} \beta_m 1(z \in R_m),$$

where $\beta_m, m = 1, \ldots, M$ denotes a constant for each region and $1(\cdot)$ denotes the indicator function.

Suppose we have n observations (Z_i, Y_i) for $i = 1, \ldots, n$. The estimated coefficients for a given partition are obtained by

minimizing the in-sample MSE:

$$\hat{\beta} = \arg \min_{b_1,\ldots,b_M} \mathbb{E}_n \left(Y - \sum_{m=1}^{M} b_m 1(Z \in R_m) \right)^2,$$

which results in

$$\hat{\beta}_m = \text{ average of } Y_i \text{ where } Z_i \in R_m.$$

The regions $R_1, \ldots, R_M$ are called nodes, and each node R_m has a predicted value $\hat{\beta}_m$ associated with it.

A nice feature of regression trees is that you get to draw cool pictures, so let's explore their usage graphically in the context of our wage example. In this example, the outcome variable Y is (log) hourly wage; and Z includes experience, geographic, and educational characteristics.

Figure 8.1 illustrates a simple regression tree for the wage data. This tree has a depth of two, meaning that predictions are produced as a sequence of two binary decisions (or partitions of the data). Starting at the top of the tree and working down provides a simple prediction rule for any observation. For example, the predicted wage for a worker without a college degree (`college` = 0) and with less than 14 years of experience (`exper` < 14) is 12 dollars an hour. We obtain this prediction by starting at the top of the tree and taking the left branch because `college` = 0 < .5. We then go left again at the second step because `exper` < 14 and arrive at the predicted value of 12.

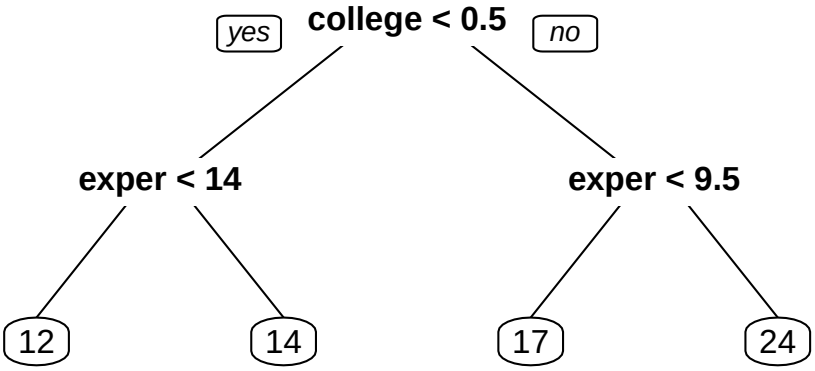


Figure 8.1: Regression tree based on wage data. The bottom nodes on the tree provide prediction rules for different subsets of observations. For example, the predicted hourly wage for a college educated worker with 9.5 or more years of experience (a worker with `college` = 1 and `exper` ≥ 9.5) is 24 dollars.

The key feature of trees is that the cut points for the partitions are adaptively chosen based on the data. That is, the splits are not pre-specified but are purely data dependent. So, how did we use the data to grow the tree in Figure 8.1?

To make computation tractable, we use recursive binary partitioning or splitting of the regressor space:

- **Growing the Tree: Level 1.** First, we cut the regressor space into two regions by choosing the regressor and splitting point such that using the prediction rule fit within each region produces the best improvement in the in-sample MSE.[1]

  Applying this procedure in the wage data gives us the depth 1 tree shown Figure 8.2. In this case, the best regressor to split on is the indicator of college degree, that takes values 0 or 1. Here splitting at any point between 0 and 1 provides the same rule, and an often used convention for binary variables is to use the "natural" split point of .5. Applying this split point yields the initial prediction rule: an hourly wage of $20 for college graduates and $13 for others.

1: To be clear, note that, in principle, finding this split point requires trying the partition produced by splitting the data along every possible value of every observed variable. That is, we are neither pre-specifying which variables nor which split points are important in providing a good prediction rule.

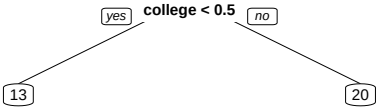


Figure 8.2: Depth 1 tree in the wage example

- **Growing the Tree: Level 2.** To grow the tree to depth 2, we then repeat the procedure for choosing the first partition rule within the two regions resulting from the first step. This step will result in a partition of the covariate space into four new regions. It is important to note that the two splits produced at this point may use different variables/splitting points than before. This feature means that the tree algorithm can create "interactions" and "nonlinearities" without requiring input from the user.

  In our example, the regions resulting from applying the first splitting rule correspond to college graduates and non-college graduates). For college graduates, the partitioning rule that minimizes in-sample MSE is to split this group into those with less than 9.5 years of experience and those with 9.5 years or more of experience. We have thus refined the prediction rule for graduates to be $24 an hour if experience is greater than or equal to 9.5 years, and $17 an hour otherwise. For non-graduates the procedure works similarly, though here the in-sample MSE minimizing split is produced by dividing non-graduates into those with less than 14 years of experience and those with 14 years of experience or more.

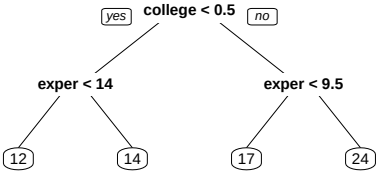


Figure 8.3: Depth 2 tree in the wage example

- **Growing the Tree: Higher Levels and Stopping Rule.** To grow deeper trees corresponding to more complex prediction rules, we simply keep repeating. We stop when the desired depth of the tree is reached,[2] or when a prespecified minimal number of observations per region, called minimal node size, is reached.

  In the wage example, we can grow a depth 3 tree by

2: One practical choice of the depth of a tree is to stop just before we get a headache from looking at a complicated tree. This rule is indeed useful if we want to present the tree as a communication device.

repeating the basic procedure within each of the four nodes of the depth 2 tree. The resulting tree is illustrated in Figure 8.4. Here, we see that the indicator for self-reported sex (`female`), high-school graduate indicator (`hsg`), and Southern region indicator (`so`) are the splitting variables chosen in the third level.

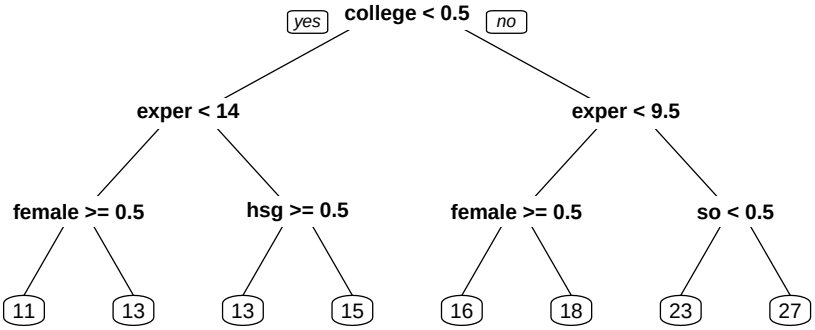


Figure 8.4: Depth 3 tree in the wage example. The depth of three was chosen to avoid getting headaches from looking at a more complicated tree.

Pruning Regression Trees. We now make several observations.

First, the deeper we grow the tree, the better is our approximation to the regression function $g(Z)$. However, the deeper the tree, the noisier our estimate $\hat{g}(Z)$ becomes, since there are fewer observations per terminal node to estimate the predicted value for this node. From a prediction point of view, we can try to find the right depth or the structure of the tree by a validation exercise such as using a single train/test split or cross-validation. For example, in the wage example, the tree of depth 2 performs better in terms of cross-validated MSE than the tree of depth 3 or 1. The process of cutting down the branches of the tree to improve predictive performance is called "Pruning the Tree."

Often for business analytics and explainability, simple trees like the ones shown are used. If we only care about building good prediction rules, we may build complicated trees and apply pruning to improve predictive performance. A simple penalty for the complexity of the tree is the number of leaves (terminal nodes) times a penalty level, where the penalty level is chosen heuristically; see, e.g, [2]. For example, we can always use a train/test split or cross-validation to settle on a penalty level. There is not a rigorously justified plug-in penalty level for trees like there is for Lasso. Figuring out such a plug-in rule is actually a good research problem.

Figure 8.5: "To prune a tree." Source: Wikipedia

Random Forests

In practice, regression trees often do not provide the best predictive performance, because a single regression tree provides a relatively crude approximation to a smooth regression function g(Z). We illustrate the potential poor approximation of regression trees in Figures 8.6 and 8.7. These figures simply illustrate that step functions, which are the outputs of typical regression tree implementations, struggle in approximating smooth functions.

A powerful and widely used approach that aims to improve upon simple regression trees is to build a random forest, as proposed by Leo Breiman [3]. The idea of a random forest is to grow many different deep trees that have low approximation error and then average the prediction rules across trees.

To produce different trees using only the observed data, the trees going into a random forest are grown from artificial data generated by sampling randomly with replacement from the original data; that is, each tree in a random forest is fit to a *bootstrap sample*.[3] Within the bootstrap samples, trees are grown deep to keep approximation error low. Averaging across the trees produced in the bootstrap samples is then meant to reduce the noisiness of the individual trees. The procedure of averaging noisy prediction rules over bootstrap samples is called Bootstrap Aggregation or Bagging. When the data set is large, we can also rely on fitting trees within *subsamples*[4] instead of using the bootstrap. Using subsamples offers some computational advantages and also simplifies theoretical analysis.

The idea seems very unusual, so let us explain again.

Each bootstrap sample is created by sampling from our data on pairs (Y_i, Z_i) randomly, with replacement. Hence, some observations are drawn multiple times and some aren't redrawn at all. Given a bootstrap sample, indexed by b, we build a tree-based prediction rule $\hat{g}_b(Z)$. We repeat the procedure B times in total, and then average the prediction rules that result from each of the bootstrap samples:

$$\hat{g}_{\text{random forest}}(Z) = \frac{1}{B}\sum_{b=1}^{B} \hat{g}_b(Z).$$

The use of the bootstrap here is unusual, yet corresponds to an intuitive idea: If we could have many independent copies of

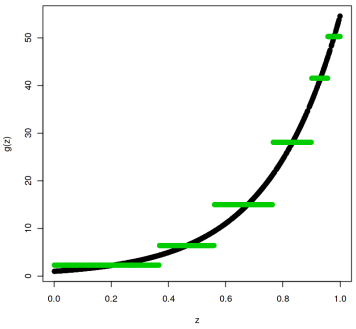


Figure 8.6: Approximation of $g(Z) = \exp(4Z)$ by a shallow regression tree in the noiseless case.

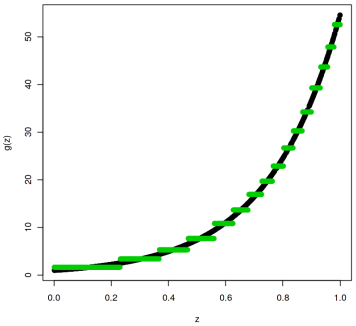


Figure 8.7: Approximation of $g(Z) = \exp(4Z)$ by a deep regression tree in the noiseless case.

3: *bootstrap sample*: typically a sample of the same or similar size to the size of the original dataset produced by sampling uniformly from the original data with replacement. Other sampling schemes may also be used, e.g. to accommodate dependence.

4: *subsample*: typically a sample of size much smaller than the original dataset produced by sampling uniformly from the original data without replacement. Other sampling schemes may also be used, e.g. to accommodate dependence.

the data, we could obtain low-bias but potentially very noisy prediction rules in each copy of the data and then average the prediction rules obtained over these copies to reduce the noise. Since we don't have many copies in reality, we rely on the bootstrap to create many quasi-copies of the data. Another feature of this idea is that the cut-points defining partitions for the tree obtained within each bootstrap sample will be different, producing a different step function approximation. Averaging over many step functions with steps at different locations will potentially produce a much smoother approximation to the underlying function. The improved approximation relative to simple trees is illustrated in Figure 8.8.

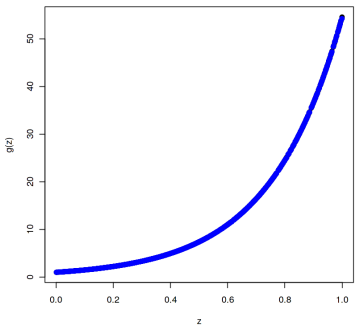


Figure 8.8: Approximation of $g(Z) = \exp(4Z)$ by a random forest in the noiseless case.

There are many modifications of the simple version of bootstrap aggregation that we have discussed. The most important modification is the use of additional randomization to "decorrelate" the trees: When we build trees over different bootstrap samples, we also randomize over the variables that trees are allowed to use in forming partitions. This additional layer of randomization encourages trees in different bootstrap samples to have different structure throughout the tree – both near the top and at the bottom – by forcing consideration of distinct sets of variables.

In summary, a random forest is an average of tree based prediction rules (a forest) produced from bootstrap or subsample data (generated randomly).

Boosted Trees

The idea of boosting is that of recursive fitting: We estimate a simple prediction rule, then take the *residuals*[5] and estimate another simple prediction rule for these residuals. We then take the residuals produced from this new prediction rules and build yet another simple model to predict them. We keep repeating this process until we reach some stopping criterion. The sum of these prediction rules fitted at each step then gives us the overall prediction rule for the outcome.

5: *residuals*: the unexplained part of an outcome we want to predict, after subtracting the prediction from the observed outcome.

Boosting can be applied with any type of base prediction rule. A common use of boosting is with regression trees which leads to *boosted trees*. Boosted trees are built up using shallow trees as the simple prediction rule. Shallow trees are trees with very few levels of depth. By keeping depth low, shallow trees produce low noise prediction rules. However, shallow trees also tend to have high approximation error because they rely on step functions with very few steps to approximate the

target regression function. That is, a single shallow regression tree tends to produce a high bias, low variance prediction rule. Boosting then helps alleviate the bias of shallow regression trees. At each step, fitting a model to the residuals from the previous step reduces the approximation error from the previous step. The improved approximation of boosted trees relative to simple trees is illustrated in Figure 8.9.

The boosting algorithm

1. Initialize the residuals: $R_i := Y_i, i = 1, \ldots, n$.
2. For $j = 1, \ldots, J$
   a) fit a tree-based prediction rule $\hat{g}_j(Z)$ to the data $(Z_i, R_i)_{i=1}^n$;
   b) update the residuals $R_i := R_i - \lambda \hat{g}_j(Z_i)$, where λ is called the learning rate.
3. Output the boosted prediction rule:

$$\hat{g}(Z) := \sum_{j=1}^{J} \lambda \hat{g}_j(Z).$$

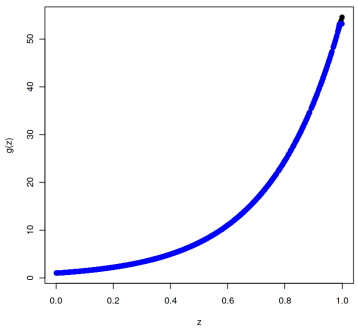


Figure 8.9: Approximation of $g(Z) = \exp(4Z)$ by boosted trees in the noiseless case with a sufficient number of steps J.

In practice, using boosted trees requires making several choices. One needs to define the tree-based prediction rule used at each step and also choose the number of learning steps, J, and the learning rate, λ. These tuning parameters are typically chosen by cross-validation.[6]

Note that the boosting algorithm is quite general and can be applied to non-tree uses. Note that the number of learning steps for boosting is important across any implementation. Because each step is building a model to predict the unexplained part of the outcome from the previous step, the in-sample prediction errors – the fit to the outcomes used to train the model – must weakly increase with each additional step. If too many iterations are taken, it is thus likely that overfitting will occur, but too few iterations may leave significant bias in the final prediction rule. In practice, the number of iterations is typically chosen by stopping the procedure once there is no marginal improvement to cross-validated MSE. A very popular implementation widely used in industry is xgboost, which has the capability to impose qualitative shape constraints like monotonicity in one or several variables. Other frequently used implementations are lightgbm and catboost.

6: We need $0 < \lambda < 1$, and a common default value for λ is 0.1. The idea of boosting is to fit simple prediction rules, so one will typically specify the prediction rule by setting the depth of the trees to a small number. For example, at each step, the prediction rule may be a regression tree of depth one (so-called stumps) or depth two. Typically, one will try several small values for depth and again choose among them by cross-validation.

8.3 Neural Nets / Deep Learning

Neural networks are a very powerful tool for modelling nonlinear relationships. They rely on many constructed regressors to approximate $g(Z)$, the conditional expectation given the regressors. The method and the name "neural networks" were loosely inspired by the mode of operation of the human brain, and developed by scientists working in Artificial Intelligence. They can be represented by cool graphs and diagrams.

Basic Ideas

First, we focus on a single layer neural network to introduce the more formal definition of neural nets. The estimated prediction rule will take the form:

$$\hat{g}(Z) := \hat{\beta}' X(\hat{\alpha}) := \sum_{m=1}^{M} \hat{\beta}_m X_m(\hat{\alpha}_m),$$

where the $X_m(\hat{\alpha}_m)$'s are constructed regressors called *neurons*,

$$\alpha = (\alpha_m)_{m=1}^{M}, \quad \beta = (\beta_m)_{m=1}^{M}, \quad X(\alpha) = (X_m(\alpha_m))_{m=1}^{M}.$$

We always take Z to include a constant of 1 as a component and set $X_1(\alpha) = 1$. The remaining neurons are generated as

$$X_m(\alpha_m) = \sigma(\alpha_m' Z), \;\; m = 2, \ldots, M,$$

where α_m's are neuron-specific vectors of parameters called weights, and σ is an activation function chosen by the practitioner. Popular activation functions are

- the sigmoid function,

$$\sigma(v) = \frac{1}{1 + e^{-v}},$$

- the rectified linear unit function (ReLU),

$$\sigma(v) = \max(0, v),$$

- the smoothed rectified linear unit function (SReLU),

$$\sigma(v) = \log(1 + \exp(v)),$$

- the leaky rectified linear unit function (Leaky-ReLU),

$$\sigma(v) = \alpha v 1(v < 0) + v 1(v \geq 0)$$

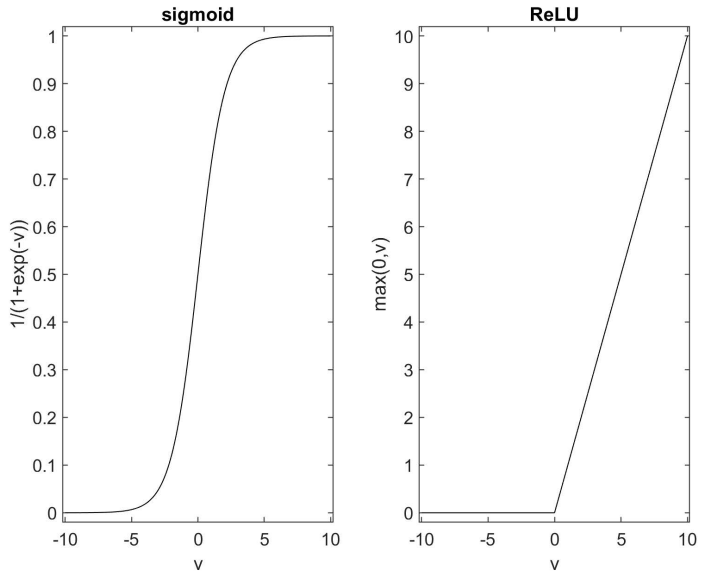


Figure 8.10: The sigmoid (logit) and ReLU activation functions

- or the linear function,

$$\sigma(v) = v.$$

The use of nonlinear activation functions is critical for generating high-quality approximations.

The estimators $\{\hat{\alpha}_m\}$ and $\{\hat{\beta}_m\}$, for $m = 1, \ldots, M$, are obtained as the solution to a penalized nonlinear least squares problem. For example, we could obtain parameter estimates by solving

$$\min_{\{\alpha_m\},\{\beta_m\}} \sum_i \left(Y_i - \sum_{m=1}^{M} \beta'_m X_{im}(\alpha_m) \right)^2 + \text{pen}(\alpha, \beta; \lambda), \quad (8.3.1)$$

where $\text{pen}(\alpha, \beta; \lambda)$ is a penalty function with penalty parameter λ. Common penalty functions are Lasso-type ℓ_1 penalties,

$$\lambda \left(\sum_m \sum_j |\alpha_{mj}| + \sum_m |\beta_m| \right),$$

and Ridge-type ℓ_2 penalties,[7]

$$\lambda \left(\sum_m \sum_j (\alpha_{mj})^2 + \sum_m (\beta_m)^2 \right).$$

Neural network estimates are typically computed using stochastic gradient descent (SGD) algorithms. In its simplest version, SGD proceeds as follows: At each step, parameters are updated

7: In many implementations of neural network training the ℓ_2 penalty is referred to as the "weight decay" parameter; inspired by the fact that the ℓ_2 penalty adds an extra $-2\lambda w$ term in the gradient calculated at each gradient step of SGD – see below – for each parameter w, with w being the parameter's current value. Thus it always "decays" the parameter towards zero.

based on the update formula

$$(\alpha, \beta) \leftarrow (\alpha, \beta) - \eta \partial_{\alpha,\beta} \text{Loss}(B; \alpha, \beta)$$

where $B \subset \{1, \ldots, n\}$ is a subset of the observations and the loss is the penalized non-linear least squares objective in Equation (8.3.1) calculated on the subset B:

$$\text{Loss}(B; \alpha, \beta) := \sum_{i \in B} \left(Y_i - \sum_{m=1}^{M} \beta'_m X_{im}(\alpha_m) \right)^2 + \text{pen}(\alpha, \beta; \lambda).$$

In other words, every time we take a small step in the direction opposite to an approximate (or stochastic) version of the gradient of the loss that we want to minimize. The gradient designates the direction of the parameters in which the loss increases the most, and the opposite is the direction in which the loss *decreases* the most.[8] The magnitude of the step is controlled by the parameter η, which is many times referred to as the *step-size*.

8: This direction is typically referred to as the direction of *steepest descent*

In SGD, gradients are computed on subsamples of data (often consisting of a single observation) called batches, and a single cycle through all subsamples is termed an "epoch." By only making use of batches of observations, SGD algorithms are able to scale to massive data sets. Using subsamples of data introduces "stochasticity" relative to using the "full" gradient computed on the entire data. In addition to the computational advantages SGD enjoys in large data sets, the presence of this noise in the computation of gradients also seems to have advantages in helping SGD algorithms avoid local saddle points. There are many fine practical details in terms of efficient computation of gradients for deep neural nets, how updating is done in SGD algorithms in general, and in the application of SGD to learning parameters of deep neural nets.[9]

9: These details are outside of the scope of this monograph. Interested readers might refer to *Deep Learning* by Goodfellow, Bengio, and Courville [4] for a textbook treatment of these issues. A popular method for training neural networks is called Adam; see this Towards Data Science blog for a detailed explanation [5].

The optimization methods employed for learning neural network parameters provide avenues for regularization beyond simply penalizing the size of the coefficients. A popular regularization method is *dropout* regularization where each neuron in a given layer can be set to zero with a given probability – for example, .1 – during parameter update steps. Dropout encourages more robust networks: If a particular neuron is important, the dropout regularization encourages creation of very similar neurons that can replicate the properties of the given neuron. Therefore, dropout regularization can be viewed as a penalty that forces similar weights for groups of neurons.

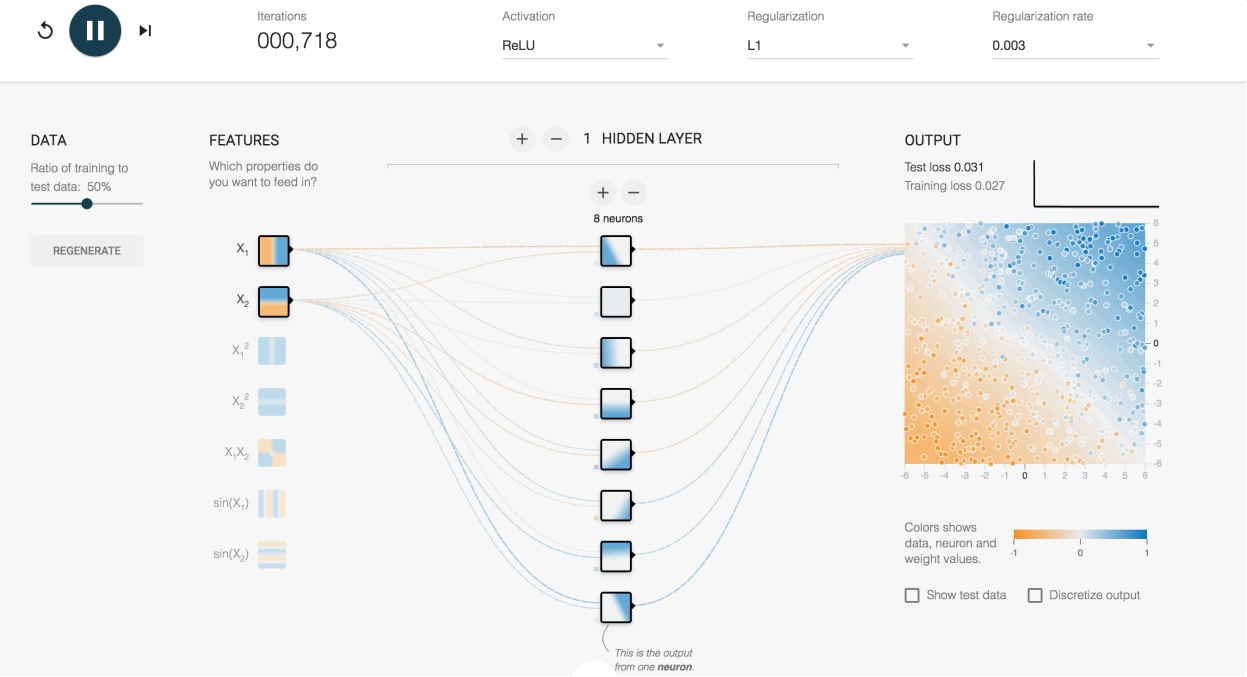
Another commonly used regularization device used with neural

networks is *early stopping*. With early stopping, a measure of out-of-sample prediction accuracy is monitored along with the value of the in-sample objective function (8.3.1). Rather than optimizing until the in-sample objective function is minimized, optimization proceeds until out-of-sample performance appears to start to degrade. By updating parameters based on in-sample fit but stopping based on out-of-sample performance, early stopping helps guard against overfitting.

As can be seen from the preceding paragraphs, using neural networks in practice relies on the choice of many tuning parameters. As there is relatively little theoretical guidance on these choices, tuning parameters are typically chosen using validation exercises such as cross-validation or simple train/test splits. An important choice that clearly relates to model flexibility is the number of neurons – the *width* of the network. We must also choose the number of layers of neurons – the *depth* of the network – in the deeper networks discussed below. Having more neurons or layers gives us additional flexibility, just like having more constructed regressors provides more flexibility in high-dimensional linear models. Other choices about regularization then interact with the choice of how many neurons and layers to use in preventing overfitting.[10]

To visualize the working of a neural network, we rely on a resource called playground.tensorflow.org [7], with which we produced a prediction regression model using a simple single layer neural network model based on two input variables. A screenshot taken after training the model is shown below.

10: There has been a flurry of recent research considering the use of very large neural networks with many more parameters than the number of observations that may easily overfit the data. These papers find that such highly overparameterized neural networks tend to find solutions that generalize well.



The network depicts the process of taking raw regressors and transforming them into predicted values. In the second column (labeled "FEATURES"), we see the inputs – our two raw regressors. The third column depicts a *hidden layer* made up of eight neurons.[11] Each neuron is constructed as a (weighted) linear combination of the raw regressors transformed by an activation function. Here we use the ReLU activation function. The neurons are connected to the inputs, and the connections represent the $\hat{\alpha}_m$ coefficients. The coloring represents the sign of the coefficients – orange is negative and blue positive – and the width of the connections represents the size of the coefficients.

11: "Hidden" refers to the fact that these layers are typically not reported. However, these layers can be extracted and used as technical regressors for other tasks. We discuss using hidden layers as features in Chapter 10 which deals with feature engineering.

Finally, the neurons are combined linearly to produce the output – the prediction rule. The connections going outwards from the neurons to the output represent the coefficients $\hat{\beta}_m$ of the linear combination of the neurons that produce the final output. The coloring and the width again represent the sign and the size of these coefficients.

The output (prediction) is shown here by the "heat" map in the box on the right. On the horizontal and vertical axes we see the values of the two inputs. The color and its intensity in the "heat" map represent the predicted value.

At the top of the screenshot, we also see that we used "L1" for the type of regularization, which corresponds to using the Lasso-type penalty. Here, the penalty level is called the regularization rate and is provided as the last entry in the top line of the screenshot.

In this example, we used a single layer neural network. If we add one or two additional layers of neurons constructed from the previous layer of neurons we get a "deep" network. We illustrate a two-layer network in the following figure.

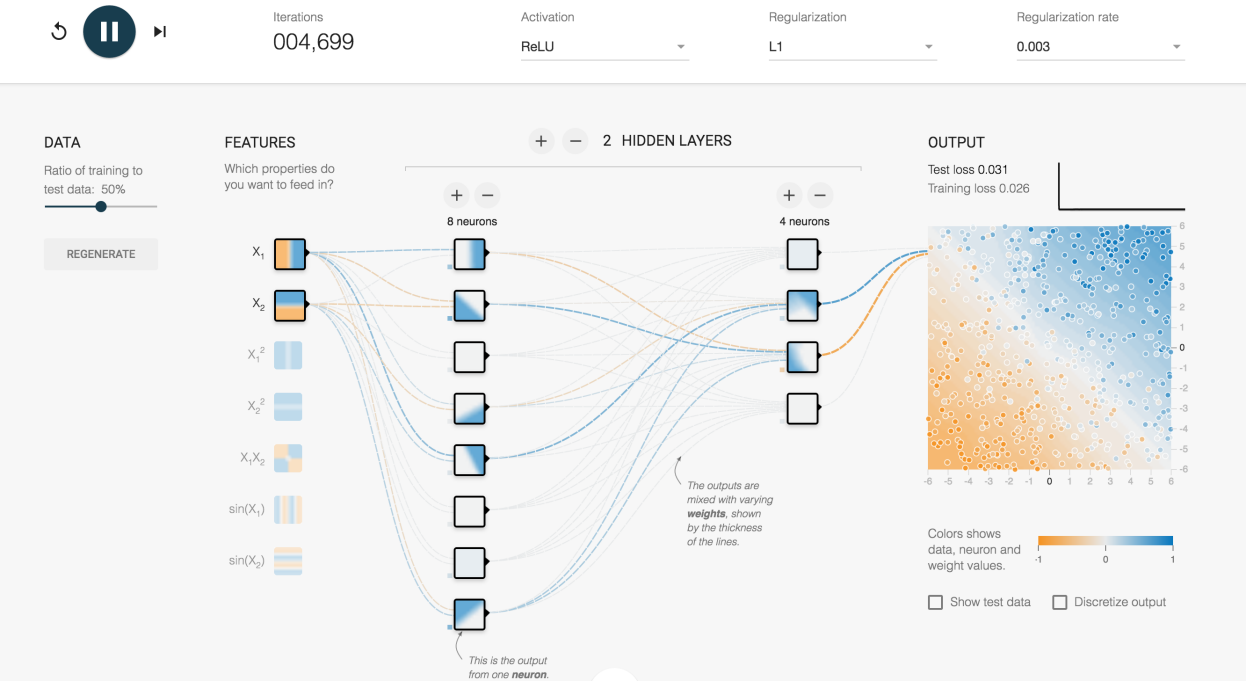
Prediction methods based on neural networks with several layers of neurons are called "deep learning" methods.

Deep Neural Networks

Here, we present the structure of a neural network with general depth. Networks with depth greater than one are called deep neural networks (DNN).

For the sake of generality, we consider networks of the multitask form, where we try to predict multiple outputs Y^t, $t = 1, ..., T$, where t stands for the "task."[12] A typical scenario is to just have one task, $T = 1$. Indeed, this scenario encompasses all of the

12: For example, we might be interested in predicting the price of a product using product characteristics across multiple markets or time periods, t. In treatment effect analysis, we may build a single neural network to predict both the outcome, Y, and the treatment, D, using other covariates. We could view this as a multitask learning problem where we are interested in two outputs, $Y^1 = Y$ and $Y^2 = D$.



discussion to this point in this text. However, there are many cases where we can use a single DNN to solve multiple tasks.

The general nonlinear regression model we work with in our discussion of deep neural networks takes the form

$$Z \overset{f_1}{\longmapsto} H^{(1)} \overset{f_2}{\longmapsto} \quad \ldots \overset{f_m}{\longmapsto} H^{(m)} \overset{f_{m+1}}{\longmapsto} \{X^t\}_{t=1}^T, \tag{8.3.2}$$

where

$$H^{(\ell)} = \{H_k^{(\ell)}\}_{k=1}^{K_\ell}$$

are called neurons, Z is the original input, and the map f_ℓ maps one layer of neurons to the next. The maps f_ℓ are defined as

$$f_\ell : v \longmapsto \{H_k^{(\ell)}(v)\}_{k=1}^{K_\ell} := (1, \{\sigma_{k,\ell}(v'\alpha_{k,\ell})\}_{k=2}^{K_\ell}), \tag{8.3.3}$$

where $\sigma_{k,\ell}$ is the activation function that can vary with the layer ℓ and across neurons k in a given layer.[13] We always include a constant of 1 as a component of Z, and we always designate one of the neurons in each layer up to m to be 1. The final layer, f_{m+1}, does not output the constant of 1 as a component:

$$f_{m+1} : v \longmapsto \{X^t(v)\}_{t=1}^T := (\{\sigma_{t,m+1}(v'\alpha_{t,\ell})\}_{t=1}^T). \tag{8.3.4}$$

13: Common architectures employ activation functions that do not vary with k. However, some popular architectures do allow activation functions to vary. For example, ResNet50, a popular image classification model that we mention in the caption to Figure 8.13, can be viewed as having activation functions that depend on k.

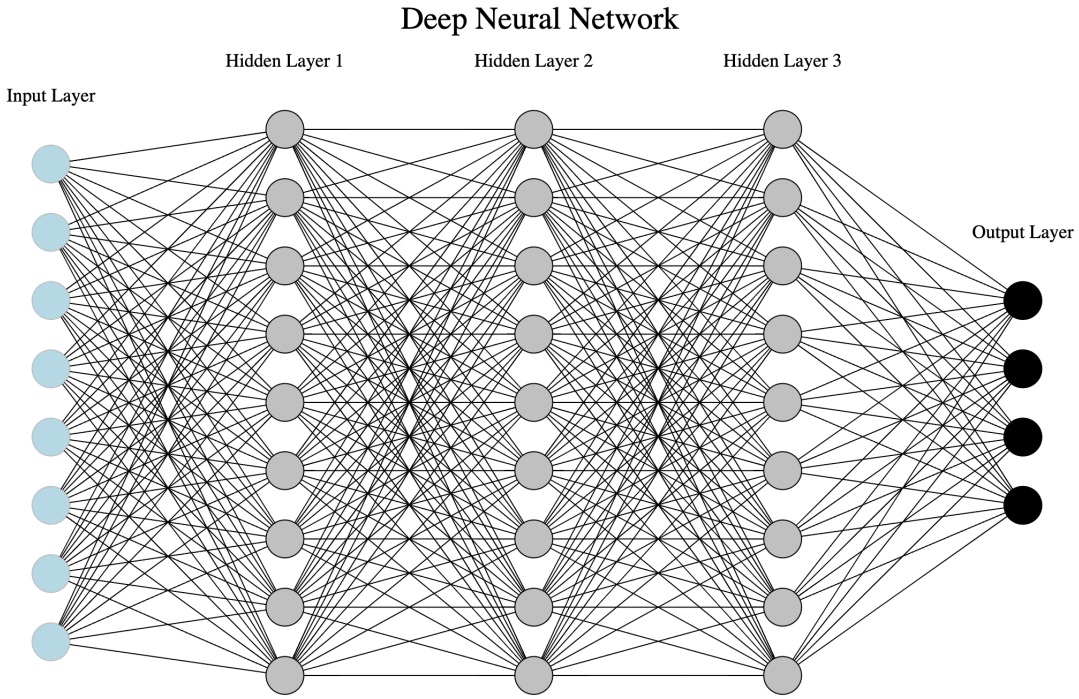


Figure 8.11: Standard Architecture of a Deep Neural Network. The input is mapped nonlinearly into the first hidden layer of the neurons. The output of this first mapping is then mapped nonlinearly into the second layer. This process is then repeated m times. The output of the penultimate layer is finally mapped (linearly or nonlinearly) into the output layer, which can have multiple outputs corresponding to different tasks.

The network mapping (8.3.2) consists of repeated composition of nonlinear mappings. This structure has been shown to be an extremely powerful tool for generating flexible functional forms which yields successful approximations in a wide range of empirical problems and is backed by approximation theory. Good approximations can be achieved by both considering sufficiently many neurons and sufficiently many layers (Yarotsky, 2017 [8]; Schmidt-Hieber, 2020 [9]; Farrell et. al, 2021 [10]; Kidger and Lyons, 2020 [11]). In empirical economic examples, it is common to just use a few hidden layers, while much deeper networks are typically used in image processing and text applications.

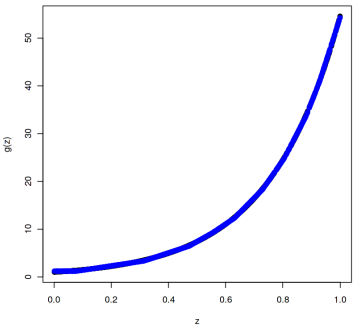


Figure 8.12: Approximation of $g(Z) = \exp(4Z)$ by a Neural Network

Similarly to single layer neural networks, the DNN model can be trained by minimizing the loss function

$$\min_{\eta \in \mathcal{N}} \sum_t w_t \sum_i (Y_i^t - X_i^t(\eta))^2 + \text{pen}(\eta; \lambda), \tag{8.3.5}$$

where η denotes all of the parameters of the mapping

$$Z_i \mapsto X_i^t(\eta),$$

w_t denotes the weight given to a task t, and $\text{pen}(\eta; \lambda)$ is a penalty function with λ denoting the penalty level.

8.4 Prediction Quality of Modern Nonlinear Regression Methods

As we have already mentioned, the best prediction rule for an outcome Y using features/regressors Z is the function $g(Z)$, equal to the conditional expectation of Y using Z:

$$g(Z) = \mathrm{E}[Y \mid Z].$$

Modern nonlinear regression methods, when appropriately tuned and under some regularity conditions, provide estimated prediction rules $\hat{g}(Z)$ that approximate the best prediction rule $g(Z)$ well.

Theoretical work demonstrates that under appropriate regularity conditions and with appropriate choices of tuning parameters, the mean squared approximation error of prediction rules produced by modern nonlinear regression methods is small once the sample size n is sufficiently large, namely,

$$\|\hat{g} - g\|_{L^2(Z)} = \sqrt{\mathrm{E}_Z[(\hat{g}(Z) - g(Z))^2]} \to 0, \quad \text{as } n \to \infty,$$

where E_Z denotes the expectation taken over Z, holding everything else fixed. To deliver these guarantees in high-dimensional settings where the number of features is large, we rely on structured assumptions, such as sparsity in the case of Lasso.

Learning Guarantees of Trees and Forests

One important property of adaptively built trees is that they are able to identify the relevant dimensions along which the regression function varies. To isolate this type of behavior of trees and forests, we consider a setting where all the regressors are binary, i.e. $Z \in \{0,1\}^d$. Considering all binary regressors is without loss of generality for categorical (discrete-valued) regressors, since each level of the regressor can be coded as a binary indicator.[14]

Without further assumptions on the regression function $g : \{0,1\}^d \to \mathbb{R}$, the best convergence rates that one could hope for scale at least at a $\sqrt{2^d/n}$ rate. Even for a moderate number of variables d, this rate of convergence can be prohibitively slow.

Adaptively built trees are particularly successful when there is only a small subset S, of size $|S| = r$, among the d variables that is relevant. Using this principle, we can formulate a nonparametric

14: Continuous regressors can also be discretized. However, discretization entails some loss of generality, and approximation properties following discretization have not been formally investigated.

analogue of the sparsity assumption that we analyzed in the case of high-dimensional linear regression with Lasso that allows us to improve on the convergence rate obtained without restrictions.

Assumption 8.4.1 (Nonparametric Sparsity of a Regression Function with Binary Regressors) *We assume that there exists a subset S of size $|S| = r$, such that the function g can be written as a function of only the variables in S; i.e. we can write*

$$g(Z) = f(Z_S)$$

where Z_S is the subvector of Z containing only the coordinates in S.

The assumption can probably be relaxed to "approximate" sparsity.[15]

15: This relaxation has not been formally investigated.

Observe that, unlike the sparsity assumption we made in the case of high-dimensional penalized linear regression, Assumption 8.4.1 imposes no restrictions on the form of the function f that takes as input the relevant variables.[16] Here, under the nonparametric sparsity assumption together with several other regularity conditions, we can prove that the mean squared approximation error of shallow regression trees or "honest" and arbitrarily deep regression forests[17] scales at a

$$\sqrt{2^r \log(d) \log(n)/n}$$

rate. Thus, the convergence rate depends strongly on the sparsity level r, while the overall number of regressors d enter only logarithmically. Moreover, even if we knew the relevant variables S, we could not hope for a rate faster than $\sqrt{2^r/n}$ since we make no further assumptions on the function f. Thus, not knowing the relevant set of regressors S adds an extra multiplicative cost on the achievable rate that only grows logarithmically with the number of regressors and the sample size. See [12] for results of similar flavor for variants of regression trees in settings beyond the binary regressor case.

16: The existence of subset S of size $|S| = r$ where $r \ll d$ does depend on the particular choice of binary representation in settings where binary variables are constructed from a set of discrete variables. For example, consider a discrete variable that encodes d different occupations. Forming d binary indicators for each type of occupation is without loss of generality. However, sparsity in this representation with $r \ll d$ requires that most occupations are the same and that there are only r occupations that differ from this otherwise common baseline. There are, of course, other ways to represent exactly the same information. For example, rather than an indicator for each observation, one could construct overlapping indicators for groups of observations that one a prior believes are similar. One should remember that dealing with general discrete variables is often challenging and requires careful thought.

17: An "honest" training approach makes use of subsampling. See Theorem 8.4.2 and the discussion immediately preceding its statement.

Theorem 8.4.1 (Learning Guarantee for Shallow Regression Trees) *Suppose that (a) the regressors are binary and the outcome variable is bounded; (b) the regression function g obeys Assumption 8.4.1; (c) regularity conditions hold that lower bound the density of the support of the distribution of covariates and upper bound the degree of variance reduction in MSE that can be achieved by features not in S as described in [13]. Then a regression tree estimator $\hat{g}$,*

where the regression tree is greedily grown based on the MSE criterion up to a depth that is at least r and at most some constant multiple of r, satisfies, for $n \geq \text{const}_P 2^r \log(d/\delta)$, with probability $1-\delta$,

$$\|\hat{g} - g\|_{L^2(Z)} \leq \text{const}_P \sigma \sqrt{\frac{2^r \log(d/\delta)\log(n)}{n}},$$

where $\sigma^2 = \mathrm{E}[(Y - g(Z))^2]$ and const_P is a constant that depends on the distribution of the data.

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In our case, a greedily grown tree optimizes over the name of regressor and splitting point that achieve the best one-step improvement in the in-sample MSE at each node.

Capping the depth of the regression tree as in Theorem 8.4.1 helps avoid overfitting, since otherwise we could potentially construct binary trees that achieve zero error on the training data and have large error out-of-sample.

An alternative to avoiding overfitting is to use an ensemble approach based on sub-sampled data. To implement an ensemble approach, we train multiple regression trees, each on a random subsample (without replacement) of the original data of size $s < n$ and average the predictions of each of these trees. To formally argue about the approximation error of such sub-sampled forests, we will require the forests to be trained in an "honest" manner.

In our setting, an honest training approach is as follows: When we train a tree on a subsample, we randomly partition the data in half and we use half of the data to find the best splits in a greedy manner, and we use the other half of the data to construct the estimates at each node of the tree. Such subsampled honest forests have been recently popularized by the work of [14]. Subsequent work of [13] showed that honest forests provably adapt to non-parametric sparsity of the regression function.

Theorem 8.4.2 (Learning Guarantee for Subsampled Honest Forests) *Suppose that (a) the regressors are binary and outcome variable is bounded; (b) the regression function g obeys Assumption 8.4.1; (c) regularity conditions hold that lower bound the density of the support of the distribution of covariates and upper bound the degree of variance reduction in MSE that can be achieved by features not in S as described in [13]. Then a regression forest estimator $\hat{g}$, where each regression tree is built in an honest manner and on a random subsample (without replacement) of size $s = \text{const}_P\, 2^r \log(d/\delta)$ of the original data, satisfies, for*

$n \geq \text{const}_P \, 2^r \, \log(d/\delta)$ with probability $1-\delta$,

$$\|\hat{g} - g\|_{L^2(Z)} \leq \text{const}_P \sigma \sqrt{\frac{2^r \, \log(d/\delta) \, \text{polylog}(n)}{n}}$$

where $\sigma^2 = \mathrm{E}[(Y - g(Z))^2]$ and const_P is a constant that depends on the distribution of the data and $\text{polylog}(n)$ is a polynomial factor of $\log(n)$.

Learning Guarantees of DNNs

We say that a function $g : \mathbb{R}^d \to \mathbb{R}$ is β-smooth if it has $\beta \geq 1$ continuous and uniformly bounded higher-order derivatives.[18] If the regression function g is only known to be β-smooth, then the best estimator of this function has estimation error, in the worst case, that converges at the rate

$$n^{-\beta/(2\beta+d)},$$

as shown by Charles Stone [15]. When d is not small, this rate of convergence is extremely slow, suggesting that learning a function in d variables is difficult if the dimension d is moderate and the target function is only known to be β-smooth.[19]

We can achieve better rates of convergence under some kind of structured sparsity or parsimony assumptions as we saw in the rates for high-dimensional linear models in Chapter 3. DNNs are able to take advantage of a nonlinear form of sparsity assumptions that we formulate below following Schmidt-Hieber [9].[20]

Assumption 8.4.2 (Structured Sparsity of Regression Function) *We assume that g is generated as a composition of $q+1$ vector-valued functions:*

$$g = f_q \circ \ldots \circ f_0$$

where the j-th function f_j

$$f_j : \mathbb{R}^{d_j} \to \mathbb{R}^{d_{j+1}},$$

has each of its d_{j+1} components β_j-smooth and depends only on t_j variables, where t_j can be much smaller than d_j.

The rate guarantee will depend on the parsimony/smoothness pairs:

$$(t_j, \beta_j), \quad j = 0, \ldots, q.$$

18: A more general definition allows β to be non-integer, but we focus on integer β for simplicity.

19: For instance, suppose that $\beta = 1$, i.e. the function is simply assumed to have a uniformly bounded first-order derivative. Moreover, suppose that we have $d = 10$ variables. Then the bound says we need n to be such that $n^{-1/12} \approx 0.1$ if we want an error of $\epsilon = 0.1$. Equivalently, the bound says we need $n \approx 10^{12} = 1$ trillion observations! If $\beta = 2$, we would only need a petty 10 million observations...

20: See also [16] for more recent theoretical developments on provable guarantees for neural networks under sparsity conditions.

For example, consider $g : \mathbb{R}^{100} \mapsto \mathbb{R}$,

$$g(x_1, x_2, x_3, x_4, ..., x_{100}) = f_1(f_{01}(x_3), f_{02}(x_2)).$$

Then

$$g_0 = f_1 \circ f_0; \quad d_0 = 100, d_1 = 2; \quad t_0 = 1, t_1 = 2.$$

Theorem 8.4.3 (Learning Guarantee for DNNs under Approximate Sparsity) *Suppose that (a) the regression function g obeys the structured sparsity assumption (Assumption 8.4.2); (b) the depth of the DNN model is proportional to* $\log n$*, (c) the width of the DNN model is no less than*

$$s \cdot \log n$$

where s is the effective dimension of the regression function g,

$$s := \max_{j=0,...,q} n^{\frac{t_j}{2\beta_j + t_j}};$$

and (d) other regularity conditions hold as specified in [9]. Then, there exists a sparse DNN estimator $\hat{g}$ with order $s \log n$ non-zero parameters such that, with probability approaching 1,

$$\|\hat{g} - g\|_{L^2(Z)} \leq \text{const}_P \sigma \sqrt{\frac{s}{n}} \text{polylog}(n),$$

where $\text{polylog}(n)$ *is a polynomial in* $\log(n)$*,* $\sigma^2 = \mathrm{E}[(Y - g(Z))^2]$*, and* const_P *is a constant that depends on the distribution of the data.*

This fundamental result is due to Schmidt-Hieber [9], where the reader may find the complete statement of regularity conditions and further technical details of the result. See also [10].

In the example above, despite the high-dimensional setting, $d = 100$, if f_{01}, f_{02}, f_{11} are β-smooth with $\beta \geq 2$, a sparse DNN is able to achieve the rate (ignoring logs):

$$\sqrt{\frac{s}{n}} = n^{-\beta/(2\beta+2)} \leq n^{-1/3}$$

where the effective dimension is

$$s = n^{\frac{2}{2\beta+2}}.$$

The Push for More Theory

Providing theory for tree-based methods and deep neural networks is currently an active area of research. Available results, such as those provided in Theorems 8.4.1-8.4.3 show that these methods can produce prediction rules that approximate best prediction rules well. However, there are still substantial gaps in our theoretical understanding of these methods.

For example, the rate guarantee for Honest Forests in Theorem 8.4.2 is the same as the rate for shallow trees in Theorem 8.4.1. This theory thus does not shed light on why random forests seem to achieve superior predictive performance over simple trees in many applications. Moreover, practical random forest algorithms tend to work well with default tuning choices. However, the theoretical results require careful alignment of tuning parameters to get good rate guarantees, and this alignment often seems inconsistent with default tuning choice in popular algorithms. The regularity conditions also require the explanatory power of the subset of the covariates that are relevant, S, to dominate the explanatory power of the irrelevant covariates.[21] This condition on signal strength is a sufficient condition, but it may not be necessary for good performance.

21: Irrelevance here only means that, given the set S of relevant covariates, the other variables do not contribute to the best prediction rule. It does not mean that the irrelevant covariates have no predictive power on their own.

8.4.3 is an interesting result and helps shed light on the power of neural networks to perform well in nonparametric settings with many variables under sensible dimension reducing structure as in Assumption 8.4.2. However, the result applies only to particular network architectures with particular activation functions and requires uniformly bounded weights (parameters). Imposing bounded weights is not typically done in practice as it is computationally challenging, and it is not clear that the other structure imposed in the theoretical construction maps well to the very deep and relatively unconstrained networks that are often employed in practice.

That is, there seem to remain substantial gaps in our theoretical understanding of the performance of tree-based algorithms and neural networks. Further exploring these properties is an ongoing, interesting area of study.

Trust but Verify

Both tree-based methods and neural networks provide powerful, flexible models that can deliver high-quality approximations of regression functions. However, the high degree of flexibility can lead to overfitting. Therefore, it is always important to verify

the performance on test data to make sure that the predictive model being used is actually a good one.

A simple verification procedure is data splitting, which can be performed in the following way:

1. We use a random subset of data for estimating/training the prediction rule.
2. We use the other part of the data to evaluate the quality of the prediction rule, recording out-of-sample mean squared error, R^2, or some other desired measure of prediction quality.

Recall that the part of the data used for estimation is called the training sample. The part of the data used for evaluation is called the testing or validation sample. We have a data sample containing observations on outcomes Y_i and features Z_i. Suppose we use n observations for training and m for testing/validation. We use the training sample to compute prediction rule $\hat{g}(Z)$. Let V denote the indices of the observations in the test sample. Then the out-of-sample/test mean squared error is

$$\text{MSE}_{test} = \frac{1}{m}\sum_{k\in V}(Y_k - \hat{g}(Z_k))^2.$$

The out-of-sample/test R^2 is

$$R^2_{test} = 1 - \frac{\text{MSE}_{test}}{\frac{1}{m}\sum_{k\in V} Y_k^2}.$$

A Simple Case Study using Wage Data

The notebooks for the wage prediction example are Notebooks 8.9.1.

We illustrate ideas using a data set of 5150 observations from the March Current Population Survey Supplement 2015. Y_i's are log wages of never-married workers living in the U.S. Z_i's include experience, education, 23 industry and 22 occupation indicators, and some other characteristics. We consider a variety of linear and nonlinear rules for predicting Y with Z.

Recall that we include a more comparison of penalized linear methods in these data in Chapter 3.

For the linear models, we estimate prediction rules of the form $\hat{g}(Z) = \hat{\beta}'X$ using X generated in two ways:

- (basic model) X consists of the 51 raw regressors in Z.
- (flexible model) X consists of 246 variables composed of the 51 raw regressor in Z, a fourth order polynomial in experience, and two-way interactions between the polynomial terms in experience and the non-experience variables in Z.

	MSE	S.E.	R^2
Least Squares (basic)	0.228	0.016	0.288
Least Squares (flexible)	0.243	0.016	0.238
Lasso	0.234	0.015	0.267
Post-Lasso	0.233	0.015	0.271
Lasso (flexible)	0.235	0.015	0.265
Post-Lasso (flexible)	0.236	0.016	0.261
Cross-Validated lasso	0.230	0.015	0.279
Cross-Validated ridge	0.236	0.015	0.262
Cross-Validated elnet	0.231	0.015	0.276
Cross-Validated lasso (flexible)	0.232	0.015	0.275
Cross-Validated ridge (flexible)	0.233	0.015	0.271
Cross-Validated elnet (flexible)	0.231	0.015	0.276
Random Forest	0.232	0.015	0.275
Boosted Trees	0.231	0.015	0.278
Pruned Tree	0.249	0.016	0.220
Neural Net (Early)	0.249	0.016	0.221

Table 8.1: Prediction Performance for the Test/Validation Sample.

We estimate $\hat{\beta}$ by linear regression/least squares and by the following penalized regression methods: Lasso and Post-Lasso with plug-in choice of λ, cross-validated Lasso, Ridge, and Elastic Net.

For the nonlinear models, we estimate prediction rules of the form $\hat{g}(Z)$ without imposing that $\hat{g}(Z) = \hat{\beta}'X$. That is, we do not assume prediction rules to be linear. We estimate the prediction models by random forests, regression trees, boosted trees, and a neural network. We use an implementation of the random forest where, at the step of growing a regression tree, we choose the best variable to split upon among $\sqrt{51}$ randomly selected variables.

Table 8.1 displays results based upon a single split of data into training and testing sets. It shows the test MSE in column 1, the standard error of the test MSE in column 2, and the test R^2 in column 3. We see that the best performing prediction rule is provided by OLS using the raw 51 regressors. The performance of the remaining procedures outside of the True, Neural Net, and Least Squares using a large set of regressors is also very similar to that of OLS with the original regressors and each other. Looking at standard errors, we see that the vast majority of methods have test MSE's that are within one standard error of the best test MSE, suggesting relatively little difference in performance across methods. The performance here is in line with what we saw in Chapter 3 where the simple OLS procedure performed very well. As discussed in Chapter 3, this finding is not surprising given the relatively small amount of data

and available variables and is one reason we recommend that simple prediction rules be considered along with more elaborate learners in many social science applications.

The outliers, in terms of performing relatively poorly, are OLS using the flexible set of covariates as well as the regression tree (Pruned Tree) and the neural net. OLS with the flexible set of predictors uses a relatively large number of variables relative to the sample size and seems likely to be overfit. On the other hand, the regression tree is not fully tuned, and the neural net is based on a simple *ad hoc* architecture choice and is also not tuned. Thus, there may be room to improve the performance of these methods.

8.5 Combining Predictions - Aggregation - Ensemble Learning

Given different prediction rules, we can choose either a single method or an aggregation of several methods as our prediction approach. An aggregated prediction is a linear combination of the basic prediction rules.

In econometrics and statistics, the procedures for combining several methods are called "model averaging" and "aggregation." In machine learning, these terms are relabeled as "ensembles" and "stacking."

Specifically, we consider an aggregated prediction rule of the form:

$$\tilde{g}(Z) = \sum_{k=1}^{K} \tilde{\alpha}_k \hat{g}_k(Z),$$

where $\hat{g}_k$'s denote the individual candidate prediction rules, potentially including a constant. The basic prediction rules are computed on the training data.

If the number of prediction rules, K, is small, we can figure out the coefficients of the optimal linear combination of the rules, $\tilde{\alpha}_k$, using test data V by simply running least squares of the outcomes in the test data on their associated predicted values:

$$\min_{(\alpha_k)_{k=1}^K} \sum_{i \in V} \left(Y_i - \sum_{k=1}^{K} \alpha_k \hat{g}_k(Z_i) \right)^2.$$

We wish to emphasize that here we are minimizing the sum of squared prediction errors in the test sample using the prediction rules from the training sample as the regressors. If K is large, we can instead use Lasso for aggregation:

$$\min_{(\alpha_k)_{k=1}^K} \sum_{i \in V} \left(Y_i - \sum_{k=1}^{K} \alpha_k \hat{g}_k(Z_i) \right)^2 + \lambda \sum_{k=1}^{K} |\alpha_k|.$$

Another popular choice is to minimize the sum of squared prediction errors in the test sample subject to the constraint that the coefficients on each candidate prediction rule are proper weights – i.e. subject to $\min_k \alpha_k \geq 0$ and $\sum_{k=1}^{K} \alpha_k = 1$.

Aggregation Results for the Case Study

We consider building an ensemble of the prediction rules fit in the wage prediction example.

	Weight OLS	Weight Lasso
Constant	-0.357	-0.097
Least Squares (basic)	1.031	0.274
Least Squares (flexible)	0.076	0.081
Lasso (basic)	1.012	0.000
Lasso (flexible)	-0.413	-0.085
Post-Lasso (basic)	0.453	0.153
Post-Lasso (flexible)	-0.237	0.000
LassoCV (basic)	-2.354	0.000
Lasso CV (flexible)	6.002	0.000
Ridge CV (basic)	0.545	0.000
Ridge CV (flexible)	0.388	0.000
ElNet CV (basic)	-0.143	0.000
ElNet CV (flexible)	-5.843	0.000
Pruned Tree	-0.091	0.000
Random Forest	0.415	0.319
Boosted Trees	0.235	0.292
Neural Net	0.048	0.000

Table 8.2: Weights of the ensemble method.

The estimated weights are shown in Table 8.2. The least squares weight estimates are relatively large in magnitude with offsetting large negative and positive values. This behavior is likely driven by the fact that the underlying prediction rules are relatively similar and thus highly correlated with each other. The weights estimated by Lasso are easier to interpret. Here we see that the majority of the weight is place on OLS with the basic set of controls, the random forest, and boosted trees. These learners seem to perform relatively well individually – see Table 8.1.

The ensemble methods also seem to mildly improve performance relative to any of the individual learners. Both the ensemble using OLS-estimated weights and the ensemble using Lasso-estimated weights have an adjusted R^2 of .30 in the test sample.

Auto ML Frameworks

There are a variety of new frameworks emerging that do automated search and aggregation of different prediction methods. These automatic aggregation procedures use approaches like the one we outlined above or other heuristics. Example implementations of automatic aggregation methods include H20, AutoML [17], Auto Gluon [18] (which relies on Neural Nets), Auto-Sklearn, Hyperopt-Sklearn and FLAML.

We've tried H20 on the wage data. It produced a model that performs similarly to OLS with the basic predictor set, yielding a test R^2 of 0.287. The performance is impressive because we gave H20 a time budget of just 100 seconds and did not specify anything other than the outcome and basic set of controls!

8.6 When Do Neural Networks Win?

The wage example may give a pessimistic impression on the power of deep learning (and machine learning more generally). A more optimistic impression emerges from examining performance of deep learning in data-rich settings, where large samples and rich features are available.

A recent example comes from Bajari et al. (2021) [19]. Here we are interested in predicting prices of products given their characteristics, which include both text and images. The resulting predictions are called hedonic prices. In this example, neural networks (specifically BERT [20] and ResNet50 [21]) are first used to convert the text and image data into several thousand-dimensional numerical features X (called embeddings). These features extracted from the text and image data are then used as input variables in a deep neural network for predicting product prices. The deep neural network used in the example consists of 3 hidden layers, with the penultimate layer consisting of about 400 neurons.

The features produced in the penultimate layer in a deep neural network are often referred to as embeddings as they encode or "embed" the information from the previous layers that is directly used in producing the final predictions. In the case of hedonic pricing, we may refer to these features as "value embeddings" as the final target is price or value of the product.

The data set used in this example is larger than 10 million observations. The accuracy of prediction for the deep neural network described above, as measured by the R^2 on the test sample, is about 90%. In contrast, random forests applied to predict prices using the text and image embeddings as inputs deliver an R^2 in the test sample that is in the ballpark of 80%, and a linear model estimated via least squares that uses the text and image embeddings as predictor variables delivers an R^2 in the test sample of only around 70%. Ignoring the neural

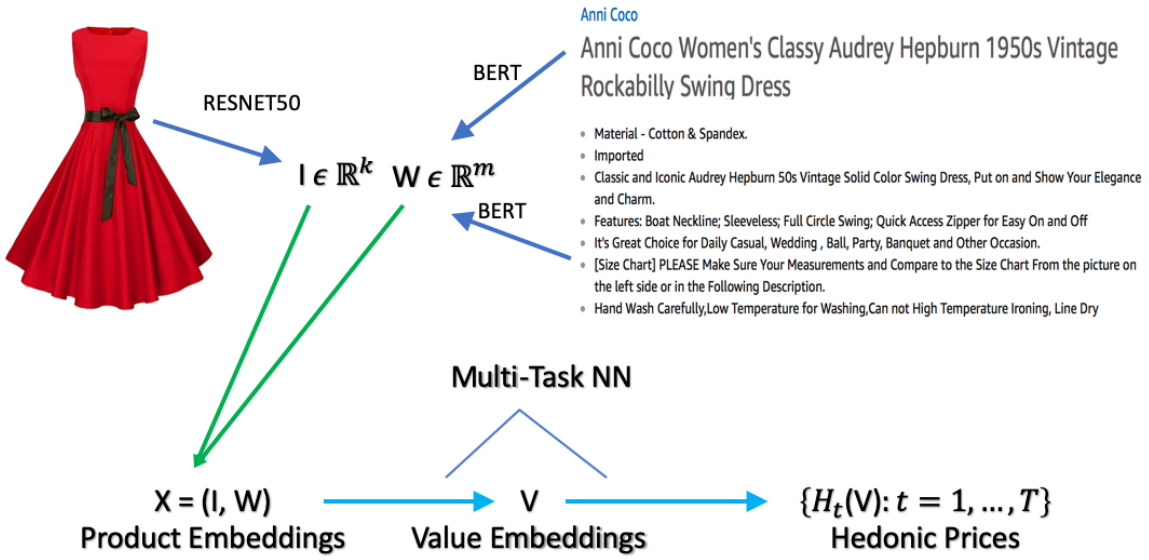


Figure 8.13: The structure of the predictive model in Bajari et al. (2021) [19]. The input consists of images and unstructured text data. The first step of the process creates numerical embeddings I and W for images and text data via deep learning methods, such as ResNet50 and BERT. The second step of the process takes as input $X = (I, W)$ and creates predictions for hedonic prices $H_t(X)$ using deep learning methods with a multi-task structure. The models of the first step are trained on tasks unrelated to predicting prices (e.g., image classification or word prediction), where embeddings are extracted as hidden layers of the neural networks. The models of the second step are trained by price prediction tasks. The multitask price prediction network creates an intermediate lower dimensional embedding $V = V(X)$, called a value embedding, and then predicts the final prices in all time periods $\{H_t(V), t = 1, ..., T\}$. Some variations of the method include fine-tuning the embeddings produced by the first step to perform well for price prediction tasks (i.e. optimizing the embedding parameters so as to minimize price prediction loss).

network embeddings of the text and image data and using only simple catalog features, the R^2 is lower than 40%.

We will discuss further details of generating embeddings in Chapter 10.

8.7 Notes

Many of the formative developments in modern nonlinear regression were led by the statistics and artificial intelligence communities. The methods were rebranded as machine learning in the 90s, and learning with neural networks was rebranded as deep learning when it was realized that deep network architectures produced phenomenal results in image classification (and later in natural language processing tasks). The success of deep neural networks was a breakthrough associated with advances in both computing power and the ability to collect very large data sets. See [22], [4], and [23] for in-depth treatments of deep learning.

In Chapter 9, we will study the use of the machine learning and deep learning for statistical inference on causal and predictive effects in high-dimensional nonlinear regression settings. In Chapter 10, we'll be using deep learning for engineering features from text and data (e.g. using images and product descriptions as "regressors").

8.8 Additional resources

- Andrej Karpathy's [24]Recipe for Training Neural Networks provides a good workflow and practical tips for training good neural network models.
- For practical details of tree-based methods, please see Hastie et al. [25] 's book "Introduction to Statistical Learning".
- For an in-depth treatment of deep learning, see Zhang et al.'s [22] "Dive Into Deep Learning", Goodfellow et al. [4] "Deep Learning", and Nielsen [23] "Neural Networks and Deep Learning".

8.9 Notebooks

Notebook 8.9.1 (ML-based Prediction of Wages) Python Notebook on ML-based Prediction of Wages and R Notebook on ML-based Prediction of Wages provide details of implementation of penalized regression, regression trees, random forest, boosted tree and neural network methods, a comparison of various methods and a way to choose the best method or create an ensemble of methods. Moreover, they provide an application of the FLAML (Python) and H2O (R) AutoML framework to the wage prediction problem. With a small time budget, both FLAML and H2O found the model that worked best for predicting wages.

Notebook 8.9.2 (Approximation of a Function by Random Forest and Neural Network) Python Notebook on Approximation of a Function by Random Forest and Neural Network and R Notebook on Approximation of a Function by Random Forest and Neural Network illustrate the flexibility of these methods in approximating the function $\exp(4x)$.

8.10 Exercises

Exercise 8.10.1 (Tree-bases Strategies) Use two paragraphs to explain to a friend how one of the tree-based strategies works.

Exercise 8.10.2 (Neural Network) Use two paragraphs to explain to a friend how a basic neural network works.

Exercise 8.10.3 (Hands-on Example I) Experiment with one of the empirical notebooks provided and summarize your findings. For example, try to see if you can build a better performing neural network in the wage example. One possibility is to use custom models in Keras, where we can construct a partially linear model that borrows the strength of the basic linear model and corrects it slightly with a nonlinear deviation function.

Exercise 8.10.4 (Hands-on Example I) Experiment with the last (non-empirical) notebook. See, for example, if you can find a (much) simpler neural network that provides the same quality of fit as the current example in the notebook.

8.A Variable Importance via Permutations

There are many ways of assessing variable importance in nonlinear models. A very simple one is the following permutation method.

The importance of variable j in any machine learning algorithm (linear or nonlinear) can be defined by computing the loss in predictive performance that results from replacing the observations of the j-th feature $(Z_{ji})_{i=1}^n$ with their random permutation

$$(Z_{j\pi(i)})_{i=1}^n,$$

where $\pi : \{1, ..., n\} \to \{1, ..., n\}$ is a permutation map, generated at random. The loss is averaged over many random permutations, to obtain an average loss measure L_j. Then the variables are ranked in terms of L_j, from largest to smallest. The top-ranked variables are taken to be the most important ones. This idea, that appeared in the original paper by L. Breiman [3], mimics the situation where the permuted regressor is an

irrelevant predictor having the same marginal distribution as the observed regressor.

Bibliography

[1] Leo Breiman. 'Statistical modeling: The two cultures'. In: *Statistical Science* 16.3 (2001), pp. 199–231 (cited on page 192).

[2] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Vol. 1. Springer Series in Statistics, New York, 2001 (cited on page 196).

[3] Leo Breiman. 'Random forests'. In: *Machine learning* 45.1 (2001), pp. 5–32 (cited on pages 197, 220).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cited on pages 202, 218, 219).

[5] Lili Jiang. *A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam)*. 2020. URL: https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c (visited on 04/03/2022) (cited on page 202).

[6] Peter L Bartlett, Andrea Montanari, and Alexander Rakhlin. 'Deep learning: a statistical viewpoint'. In: *Acta Numerica* 30 (2021), pp. 87–201 (cited on page 203).

[7] *playground.tensorflow.org*. https://playground.tensorflow.org/. Accessed: 2022-04-03 (cited on page 203).

[8] Dmitry Yarotsky. 'Error bounds for approximations with deep ReLU networks'. In: *Neural Networks* 94 (2017), pp. 103–114 (cited on page 206).

[9] Johannes Schmidt-Hieber. 'Nonparametric regression using deep neural networks with ReLU activation function'. In: *Annals of Statistics* 48.4 (2020), pp. 1875–1897 (cited on pages 206, 210, 211).

[10] Max H. Farrell, Tengyuan Liang, and Sanjog Misra. 'Deep Neural Networks for Estimation and Inference'. In: *Econometrica* 89.1 (2021), pp. 181–213 (cited on pages 206, 211).

[11] Patrick Kidger and Terry Lyons. 'Universal Approximation with Deep Narrow Networks'. In: *Proceedings of Thirty Third Conference on Learning Theory*. Ed. by Jacob Abernethy and Shivani Agarwal. Vol. 125. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2306–2327 (cited on page 206).

[12] Stefan Wager and Guenther Walther. 'Adaptive concentration of regression trees, with application to random forests'. In: *arXiv preprint arXiv:1503.06388* (2015) (cited on page 208).

[13] Vasilis Syrgkanis and Manolis Zampetakis. 'Estimation and Inference with Trees and Forests in High Dimensions'. In: *Proceedings of Thirty Third Conference on Learning Theory*. Ed. by Jacob Abernethy and Shivani Agarwal. Vol. 125. Proceedings of Machine Learning Research. PMLR, 2020, pp. 3453–3454 (cited on pages 208, 209).

[14] Stefan Wager and Susan Athey. 'Estimation and Inference of Heterogeneous Treatment Effects using Random Forests'. In: *Journal of the American Statistical Association* 113.523 (2018), pp. 1228–1242 (cited on page 209).

[15] Charles J. Stone. 'Optimal global rates of convergence for nonparametric regression'. In: *Annals of Statistics* 10.4 (1982), pp. 1040–1053 (cited on page 210).

[16] Rahul Parhi and Robert D Nowak. 'Deep Learning Meets Sparse Regularization: A Signal Processing Perspective'. In: *arXiv preprint arXiv:2301.09554* (2023) (cited on page 210).

[17] Erin LeDell and Sebastien Poirier. 'H2O AutoML: Scalable automatic machine learning'. In: *Proceedings of the AutoML Workshop at ICML*. Vol. 2020. 2020 (cited on page 217).

[18] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 'Autogluon-tabular: Robust and accurate AutoML for structured data'. In: *arXiv preprint arXiv:2003.06505* (2020) (cited on page 217).

[19] Patrick L. Bajari, Zhihao Cen, Victor Chernozhukov, Manoj Manukonda, Jin Wang, Ramon Huerta, Junbo Li, Ling Leng, George Monokroussos, Suhas Vijaykunar, et al. *Hedonic prices and quality adjusted price indices powered by AI*. Tech. rep. cemmap working paper CWP04/21, 2021 (cited on pages 217, 218).

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 'BERT: Pre-training of deep bidirectional transformers for language understanding'. In: *arXiv preprint arXiv:1810.04805* (2018) (cited on page 217).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 'Deep residual learning for image recognition'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cited on page 217).

[22] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 'Dive into deep learning'. In: *URL: https://d2l. ai* (2020) (cited on pages 218, 219).

[23] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cited on pages 218, 219).

[24] Andrej Karpathy. *A Recipe for Training Neural Networks*. 2019. URL: http://karpathy.github.io/2019/04/25/recipe/ (visited on 04/06/2022) (cited on page 219).

[25] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*. Springer, 2013 (cited on page 219).