NEURAL NETWORKS

Alexander Quispe
The World Bank, PUCP
aw.quispero@up.edu.pe

June 24, 2024

Table of Contents

Introduction

- Neural networks are powerful for modeling non-linear relationships
- They assemble many regressors to estimate $g(Z) = \mathbb{E}[Y|Z]$
- The name ”neural network” comes from them being analogous to biological neurons

Introduction

Neural networks consist of several interconnected layers, as seen in the image below. Each node is known as a neuron. A more formal definition for neurons is presented in the next slide.

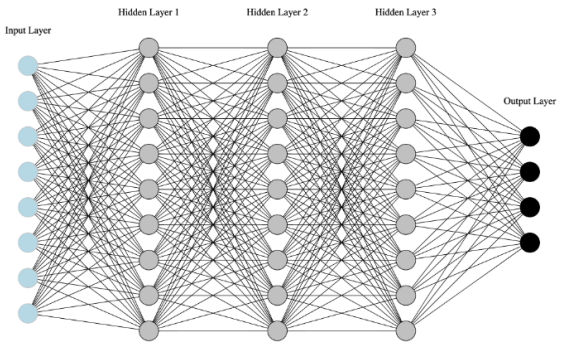

Figure: Standard Neural Network Architecture. Taken from Chernozhukov et al. (2024)

Neural Networks Basics

To simplify things for now, we focus on a single layer network. This layer contains M regressors $X_m(\alpha_m)$, called *neurons*. The estimated prediction is:

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

Here, $\hat{\alpha}_m$ is called a *weight*.

Neural Networks Basics

We assume Z to have a constant term and set $X_1(\alpha) = 1$. The other neurons take the form:

$$X_m(\alpha_m) = \sigma(\alpha'_m Z)$$

The function σ is called an *activation function*, and it is chosen by the practitioner. This function could be linear

$$\sigma(v) = v$$

but non-linear functions work much better for generating good approximations

Popular activation functions - Sigmoid

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

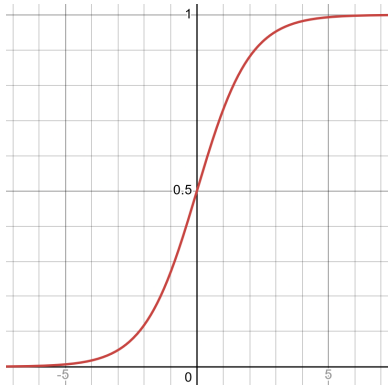

Figure: Sigmoid function

Popular activation functions - Rectified Linear Unit (ReLU)

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

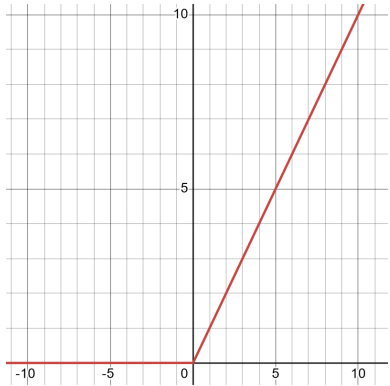

Figure: ReLU function

Popular activation functions - Smoothed ReLU

$$\sigma(v) = \log(1 + e^v)$$

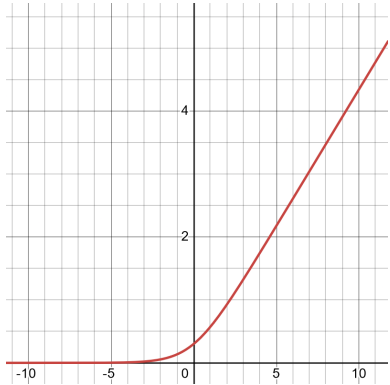

Figure: SReLU function

Popular activation functions - Leaky-ReLU

$$\sigma(v) = \epsilon v \mathbb{1}(v < 0) + v \mathbb{1}(v \geq 1)$$
$$\epsilon \in (0, 1)$$

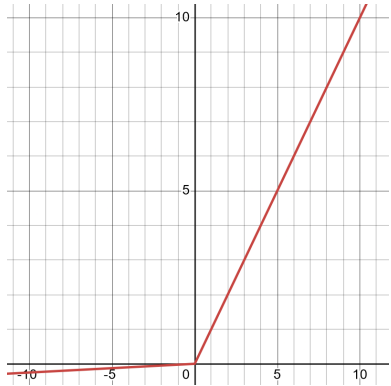

Figure: Leaky-ReLU function

Estimation of NN

The estimators $\hat{\alpha}$ and $\hat{\beta}$ are computed by solving:

$$\min_{\alpha,\beta} \sum_i \left(Y_i - \sum_{m=1}^{M} \beta_m X_m(\alpha_m) \right)^2 + \phi(\alpha, \beta; \lambda) \tag{1}$$

Where $\phi(\alpha, \beta; \lambda)$ is a penalty function with penalty parameter λ. It can be lasso-type ℓ_1

$$\lambda \left(\sum_m ||\alpha_m||_1 + ||\beta||_1 \right)$$

or ridge-type ℓ_2

$$\lambda \left(\sum_m ||\alpha_m||_2^2 + ||\beta||_2^2 \right)$$

Estimation of NN

In practice, the estimation of the parameters is done through some *stochastic gradient descent (SGD)* algorithm. The basic principle for them is the following: at each step, we update $(\hat{\alpha}, \hat{\beta})$ as follows:

$$(\hat{\alpha}, \hat{\beta}) \leftarrow (\hat{\alpha}, \hat{\beta}) - \eta \nabla C(\hat{\alpha}, \hat{\beta}; B)$$

Where $B \subset \{1, ..., n\}$ is a subset of the samples and $C(\hat{\alpha}, \hat{\beta}; B)$ is the objective function in (1), but only evaluated in the subset B

$$\sum_{i \in B} \left(Y_i - \sum_{m=1}^{M} \beta_m X_M(\alpha_m) \right)^2 + \phi(\alpha, \beta; \lambda)$$

Estimation of NN

Intuitively, this means that we take a small step (of magnitude η) in the direction opposite to the largest increase in our objective function

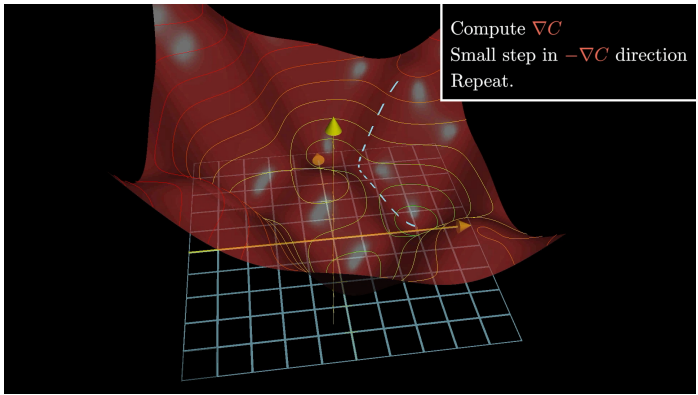

Figure: Gradient descent algorithm visualization by Grant Sanderson

Estimation of NN

Some technical details:

- The gradient $\nabla C(\hat{\alpha}, \beta; \lambda)$ is computed on the subsample B. This called a **batch**.
- Using batches is done for two reasons
  - It reduces memory requirements and computational load
  - It adds some randomness to the relative to using the full sample, which can lead to avoiding local optima.
- The magnitude of the step η is called **step size** or **learning rate**
- Training consists on cycling through all batches with SGD several times. Each cycle is called an **epoch**

Other regularization methods

Other regularization methods to improve out-of-sample performance can be used

- **Droput**
  - Consists on giving each neuron in a given layer a probability of being set to 0
  - Increases robustness of the regressors, as the NN spreads the importance of neurons
- **Early stopping**
  - For each epoch, out-of-sample inference metrics are also computed
  - The minimization of the loss stops when the out-of-sample performance degrades for a given number of epochs
  - Helps guard against overfitting

Example

The following is an example as from the Tensorflow Playground

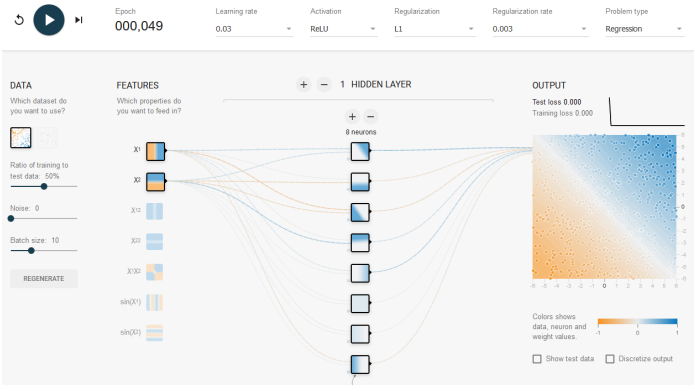

Figure: Enter Caption

Example

The Network in the previous slide has the following components:

1. The parameters of the mode: learning rate, ReLU activation and L1 regularization with $\lambda = 0.003$
2. An input layer. In this case, only two feature: X_1 and X_2
3. One hidden layer with 8 neurons.
   - Each neuron is constructed as a (weighted) linear combination of the raw regressors transformed by an activation function.
   - The connections from input features to neurons represent the $\hat{\alpha}_m$ coefficients
4. For the output, the neurons are combined linearly, with weights $\hat{\beta}$

Deep Neural Networks

Now we can take generalize for a network with more than one layer, which is called a **deep neural network** (DNN). Generally, NN's can be multitasking, which means they can have several outputs $Y^t, t = 1, \dots T$. Our nonlinear regression model with m hidden layers takes the form:

$$Z \overset{f_1}{\mapsto} H^{(1)} \overset{f_2}{\mapsto} \dots \overset{f_m}{\mapsto} H^{(m)} \overset{f_{m+1}}{\longmapsto} {X^t}_{t=1}^{T} \tag{2}$$

Z is the original input. Each $H^{(\ell)}$ is the layer ℓ which is the collection of K_ℓ neurons $H_k^{(\ell)}$:

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

Deep Neural Networks

On the other hand, the maps $f_\ell, \ell = 1, ..., m$ are defined as:

$$f_\ell : v \mapsto \{H_k^{(\ell)}(v)\}_{k=1}^{K} f := (1, \{\sigma_k^{(\ell)}(v'\alpha_k^{(\ell)})\}_{k=2}^{K_\ell}) \tag{3}$$

where $\sigma_k^{(\ell)}$ is the activation function that can vary depending on the layer ℓ and the neuron k. As we can see, for $k = 1$, the value of the neuron is 1 for all m hidden layers, and we assume that Z has an intercept column too. The final map f_{m+1} is not set to include a constant value, and it is defined by:

$$f_{m+1} : v \mapsto \{X^t(v)\}_{t=1}^{T} := (\{\sigma_t^{(m+1)}(v'\alpha_t^{(m+1)})\}_{t=1}^{T}) \tag{4}$$

Deep Neural Networks Estimation

For estimation, the minimization problem is now:

$$\min_{\eta \in \mathcal{N}} \sum_t w_t \sum_i \left(Y_i^t - X_i^t(\eta) \right)^2 + \phi(\eta; \lambda), \tag{5}$$

where η denotes all the parameters off all the mappings from input to output

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

and w_t denotes the weight given to a task t. Like before, this tends to be conducted in batches.

Convergence of estimators

For the most part, the theoretical results for modern nonlinear regression methods (like NN's or Random Forests) is that, under appropriate regularity conditions, the prediction error tends to be small with a large enough sample size n

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

However, the convergence rate might be slow, so it might still be better to use simpler methods like Lasso. We will now examine convergence rates for NN's.

Convergence of estimators

First, a function $g : \mathbb{R}^d \to \mathbb{R}$ is said to be β-smooth if it has $\beta \geq 1$ continuous and uniformly bounded higher order derivatives. If a regression g is only known to be β-smooth, the worst-case convergence rate for the estimation error is

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

This convergence rate can be extremely slow if the ratio between d and β is large. For example, if we have a function only known to have $\beta = 1$ and we have $d = 10$ variables, and we wish to have an error of $\epsilon = 0.1$, then we would need a sample size n such that

$$n^{-12} \approx 0.1$$
$$n \approx 10^{12}$$

which is 1 trillion; likely unachievable in practice. If $\beta = 2$, we the necessary size would now be 10 million

Convergence of estimators

Above result might seem discouraging. However, with some sparsity assumptions, the theoretical convergence rate speeds up.

Assumption 1 (Structured Sparsity of Regression Function)

We assume that g is a composition of $q + 1$ vector-valued functions:

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

where

$$f_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$$

has each of its d_{i+1} components β_i-smooth and only actually depends on t_i variables, where t_i can be much smaller than d_i.

If this assumption holds, the worst-case convergence rate will now only depend on (t_i, β_i)

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = ?$$
$$d_1 = ?$$
$$t_0 = ?$$
$$t_1 = ?$$

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = ?$$
$$d_1 = ?$$
$$t_0 = ?$$
$$t_1 = ?$$

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = 100$$
$$d_1 = ?$$
$$t_0 = ?$$
$$t_1 = ?$$

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = 100$$
$$d_1 = 2$$
$$t_0 = ?$$
$$t_1 = ?$$

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = 100$$
$$d_1 = 2$$
$$t_0 = 1$$
$$t_1 = ?$$

Sparsity Assumption Example

If we have the following functions

$$g(x_1, ..., x_{100}) = f_1(f_0(x_1, ..., x_{100})$$
$$f_0 = (f_{0,0}(x_2), f_{0,1}(x_3))'$$

then we have

$$d_0 = 100$$
$$d_1 = 2$$
$$t_0 = 1$$
$$t_1 = 2$$

Convergence of DNNs under Approximate Sparcity

Theorem 1 (Learning guarantee for DNNs under Approximate Sparcity)

Suppose that (1) g obeys Assumption 1; (2) the depth of the DNN is proportional to $\log n$; (3) the width of the DNN is no less than $s \cdot \log n$, where

$$s := \max_{i=0,\ldots,q} n^{\frac{t_i}{2\beta_i + t_i}}$$

is the effective dimension of g; and (4) other regularity conditions specified in Schmidt-Hieber (2020) hold. Then there exists a sparse DNN estimator $\hat{g}$

$$||\hat{g} - g||_{L^2(Z)} \leq c\sigma\sqrt{\frac{s}{n}}P(\log n)$$

Where P is a polinomial function, c is a constant that depends on the distribution of the data, and $\sigma = \mathbb{E}[Y - g(Z))^2]$

For further detail, refer to Scmidt-Hieber (2020).

Convergence of DNN under Approximate Sparcity

Our term of interest in this theorem is

$$\frac{s}{n},$$

which approximates our convergence rate well enough. In the previous example, although $d = 100$, if $f_{0,1}$, $f_{0,2}$, $f_{1,1}$ are β-smooth with $\beta \geq 2$, a sparse DNN can achieve a convergence rate of, at worst,

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

where the effective dimension is

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

This means that, if we desire an error rate of $\epsilon = 0.01$, now our approximated sample size would only be $n \approx 1000$, which is not uncommon to find in practice.

Model performance in practice

The quality of our model can be evaluated by splitting the sample into training and validation sets. We can calculate the validation MSE, R^2, etc. of our estimators. For a validation set V, the out-of-sample MSE is

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

The out-of-sample R^2 is:

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

Example with wage data

Chernozhukov et al. (2024) conduct an example of estimation and evaluation with several models with data from the March Current Population Survey Supplement 2015 dataset, which can be found in this link.
The resulting models' performance is summarized in the following table:

	MSE	S.E.	R^2
Least Squares (basic)	0.229	0.016	0.282
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	0.229	0.015	0.282
Cross-Validated Ridge	0.234	0.015	0.267
Cross-Validated Elastic Net	0.230	0.015	0.280
Cross-Validated Lasso (flexible)	0.232	0.015	0.275
Cross-Validated Ridge (flexible)	0.233	0.015	0.271
Cross-Validated Elastic Net (flexible)	0.231	0.015	0.276
Random Forest	0.233	0.015	0.270
Boosted Trees	0.230	0.015	0.279
Pruned Tree	0.248	0.016	0.224
Neural Net	0.276	0.012	0.148

Figure: Performance results of wage predictors by Chernozhukov et al. (2024)

We can see that the Neural Net does not perform as well as the other models. When do they perform better?

Best context for NNs

As exemplified by Bajari et al. (2021), neural networks perform better in context with very large amounts of observations and features. Their dataset for price prediction consists of text embeddings, which are vectors, in this case of around a thousand dimensions, that encode the meaning of text. This dataset contains more than 10 million observations. Out-of-sample performance can be summarized as follows:

- Linear models had an R^2 of about 0.7.
- Random forests had an R^2 of about 0.8.
- Neural networks had an R^2 of about 0.9.