ECON 950 — Winter 2020
Prof. James MacKinnon

11. Neural Networks

Neural networks go back many decades, but they have recently become a very hot topic because of major improvements in performance.

ESL says that the most widely used neural network model is the **single hidden layer back-propagation network**, or **single layer perceptron**.

This is no longer true. In recent years, **deep learning** has taken off, and it involves a great many hidden layers.

One of the things that held back progress for decades was the paper

Kurt Hornik, Maxwell Stinchcombe, and Halbert White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, 2 (5), 1989, 359–366.

This paper, with over 18 thousand citations, was widely believed to say that neural networks only need one hidden layer.

Abstract:

> This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.

For regression, there is typically one output Y at the top of the network diagram.

For classification, there are typically K of them, denoted Y_k.

At the bottom are p inputs.

In between are M **activation functions** which explain **derived features** Z_m, K **target functions** that map from the Z_m to T_k, and K **output functions** $g_k(\boldsymbol{T})$ which map from the T_k to the Y_k.

The activation functions for the derived features are

$$Z_m = \sigma(\alpha_{0m} + \boldsymbol{x}^\top \boldsymbol{\alpha}_m), \tag{1}$$

where the choice of $\sigma(\cdot)$ has changed over time.

For many years, the most popular choice for the activation function was the **sigmoid function**, which we call the logistic function:

$$\sigma(x) = \frac{1}{1+\exp(-x)} = \frac{\exp(x)}{1+\exp(x)}\,. \tag{2}$$

The target function that aggregates the Z_m is

$$T_k = \beta_{0k} + \boldsymbol{z}\boldsymbol{\beta}_k \tag{3}$$

The output function is typically the identity for a regression, so that $g_k(\boldsymbol{T}) = T_k$.

For classification, it is more common to use the **softmax function**

$$g_k(\boldsymbol{T}) = \frac{\exp(T_k)}{\sum_{\ell=1}^{K}\exp(T_\ell)}, \tag{4}$$

which is just the transformation used for multinomial logit.

Combining the activation functions with the output function, we obtain fitted values $f_k(\boldsymbol{x}) = g_k(\boldsymbol{T})$ via (1), (3), and (4).

Because we do not observe the Z_m, the units that compute them are called **hidden units**. There can be more than one layer of these.

If we think of the Z_m as basis expansions of the original inputs, a neural network is like a linear or multilogit model that uses them as inputs.

But, unlike basis expansions, parameters of the activation functions are estimated.

For any sigmoid function, we can scale and/or recenter the input. Evidently, $\sigma(x/2)$ rises more slowly than $\sigma(x)$, and $\sigma(2x)$ rises faster.

If we change the function from $\sigma(x)$ to $\sigma(x - x_0)$, we shift the threshold where $\sigma > 0.5$ from 0 to x_0.

If $||\boldsymbol{\alpha}||$ is very small, the sigmoid function will be almost linear.

If $||\boldsymbol{\alpha}||$ is very large, the sigmoid function will be very flat near 0, then very steep, then very flat near 1.

The neural network model with one hidden layer has the same form as the projection pursuit regression model. The difference is that the activation functions have a particular functional form.

Recall that the PPR can be written as

$$y = \sum_{m=1}^{M} g_m(\boldsymbol{x}^\top \boldsymbol{\omega}_m), \tag{5}$$

Suppose that

$$\begin{aligned} g_m(\boldsymbol{x}^\top \boldsymbol{\omega}_m) &= \beta_m \sigma(\alpha_{0m} + \boldsymbol{x}^\top \boldsymbol{\alpha}_m) \\ &= \beta_m \sigma(\alpha_{0m} + ||\alpha_m|| \boldsymbol{x}^\top \boldsymbol{\omega}_m), \end{aligned} \tag{6}$$

where $\boldsymbol{\omega} = \boldsymbol{\alpha}_m / ||\alpha_m||$ is a unit vector.

Evidently (6) is a very special case of the ridge function $g_m(\boldsymbol{x}^\top \boldsymbol{\omega}_m)$.

Because the activation functions in neural nets are much more restrictive than ridge functions in PPR, we tend to need a lot more of them.

For some years, the **hyperbolic tangent** or **tanh** function was popular as the activation function. Recall that

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{7}$$

While the logistic function ranges from 0 to 1, $\tanh(x)$ ranges from -1 to 1.

Both the logistic and tanh functions seem natural, because they map smoothly from the real line to an interval. However, they turned out to have important deficiencies.

- They both "saturate". When the argument is small, the logistic will be close to 0, and when the argument is large, it will be close to 1.
- Changing the weights (the $\boldsymbol{\alpha}_m$ vectors) has little effect when the functions are saturated.
- This is closely related to the "vanishing gradient problem." When the activation function is saturated, the gradients are very small, so it is hard to know how to vary the weights.

These problems tend to be especially severe for models with several layers. If saturation occurs for any layer, making changes to the weights for lower layers will have little impact on the model fit.

In econometric terms, identification becomes extremely difficult.

The solution is to use the **rectified linear activation unit**, or **ReLU** as the activation function.

This function is simply

$$g(x) = \max(0, x), \tag{8}$$

which is absurdly easy to calculate. It saturates if the argument is negative, but not if it is positive. In the latter case, the gradient never vanishes.

The ReLU now seems to be the default activation function for most types of neural networks.

However, there can be problems when $x < 0$. Therefore, it is generally good to start with positive inputs.

The ReLU can also be generalized in various ways. For example, the **leaky ReLU** is

$$g(x) = \mathbb{I}(x > 0)x + 0.01\mathbb{I}(x \leq 0)x. \tag{9}$$

So instead of being 0 when x is negative, it is a small negative number that has a small gradient.

There are many other generalizations, including the **exponential linear unit**:

$$g(x) = \mathbb{I}(x > 0)x + a\mathbb{I}(x \leq 0)(e^x - 1). \tag{10}$$

where a is a hyperparameter to be tuned.

11.1. Fitting Neural Networks

Neural networks generally have a lot of unknown parameters, often called **weights**. The complete set is the vector $\boldsymbol{\theta}$. It consists of

$$\alpha_{0m} \text{ and } \boldsymbol{\alpha}_m, \;\; m = 1, \ldots, M \qquad [M(p+1)] \tag{11}$$

for the activation functions, plus

$$\beta_{0k} \text{ and } \boldsymbol{\beta}_k, \;\; k = 1, \ldots, K \qquad [K(M+1)] \tag{12}$$

for the target functions.

For regression, the objective function is

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{N} R_i(\boldsymbol{\theta}) = \sum_{k=1}^{K}\sum_{i=1}^{N}\big(y_{ik} - f_k(\boldsymbol{x}_i)\big)^2. \tag{13}$$

Here, following ESL, we allow there to be more than one output, although it seems odd that there is no allowance for these to be correlated.

For classification, a sensible objective function is the deviance:

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{N} R_i(\boldsymbol{\theta}) = \sum_{k=1}^{K}\sum_{i=1}^{N} y_{ik} \log f_k(\boldsymbol{x}_i). \tag{14}$$

The corresponding classifier for any $\boldsymbol{x}$ is the value of k that maximizes $f_k(\boldsymbol{x})$.

With the softmax activation function (4), minimizing (14) is equivalent to estimating a linear logistic regression in the hidden units.

If we simply minimize (13) or (14), we are likely to overfit, perhaps severely.

The obvious solution is to regularize, but that does not seem to be what neural net folks do, perhaps because there are too many parameters.

Instead, they stop the algorithm early, before actually getting to the minimum. This involves using a validation sample as estimation progresses.

However, this means that starting values are important. With ReLU, it would be really bad to start at a point where a lot of the activation functions equal 0.

Minimizing $R(\boldsymbol{\theta})$ can be done by **back-propagation**, which is a two-pass procedure.

Starting values are often chosen randomly.

Back-propagation often works well, especially on parallel computers, because each hidden unit passes information only to and from units with which it is connected.

However, back-propagation can be slow, and better methods are available.

For the regression case,

$$R_i(\boldsymbol{\theta}) = \sum_{k=1}^{K} \big(y_{ik} - f_k(\boldsymbol{x}_i)\big)^2. \tag{15}$$

The derivatives with respect to the β_{km} are

$$\frac{\partial R_i}{\partial \beta_{km}} = -2\big(y_{ik} - f_k(\boldsymbol{x}_i)\big)\boldsymbol{g}'_k(\boldsymbol{z}_i^\top \boldsymbol{\beta}_k) z_{mi}, \tag{16}$$

where $z_{mi} = \sigma(\alpha_0 + \boldsymbol{x}_i^\top \boldsymbol{\alpha}_m)$, and $\boldsymbol{z}_i$ is an M-vector with typical element z_{mi}.

The derivatives with respect to the $\alpha_{m\ell}$ are

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -2\sum_{k=1}^{K}\big(y_{ik} - f_k(\boldsymbol{x}_i)\big)\boldsymbol{g}'_k(\boldsymbol{z}_i^\top \boldsymbol{\beta}_k)\beta_{km}\boldsymbol{\sigma}'(\boldsymbol{x}_i^\top \boldsymbol{\alpha}_m) x_{i\ell}. \tag{17}$$

A **gradient descent** update at iteration $j+1$ has the form

$$\begin{aligned} \beta_{km}^{(j+1)} &= \beta_{km}^{(j)} - \gamma_j \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(j)}} \\ \alpha_{m\ell}^{(j+1)} &= \alpha_{m\ell}^{(j)} - \gamma_j \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(j)}} \end{aligned}, \tag{18}$$

where γ_j is the **learning rate**.

We can rewrite the derivatives in (16) and (17) as

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi} \tag{19}$$

and

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = s_{mi} x_{i\ell}. \tag{20}$$

For example,

$$\delta_{ki} = -2\big(y_{ik} - f_k(\boldsymbol{x}_i)\big)\boldsymbol{g}_k'(\boldsymbol{z}_i^\top \boldsymbol{\beta}_k), \tag{21}$$

and we can see from (17) that s_{mi} is even more complicated.

We can think of δ_{ki} and s_{mi} as “errors” from the current model at the output and hidden layer unit, respectively.

These errors satisfy the **back-propagation equations**

$$s_{mi} = \sigma'(\boldsymbol{x}_i^\top \boldsymbol{\alpha}_m) \sum_{k=1}^{K} \beta_{km}\delta_{ki}. \tag{22}$$

In the **forward pass**, the current parameters are fixed and the predicted values $\hat{f}_k(\boldsymbol{x}_i)$ are computed using the activation and output functions.

In the **backward pass**, the δ_{ki} are computed and then back-propagated using (22) to give the s_{mi}.

Both the δ_{ki} and the s_{mi} are then used to compute the gradients for the updates in (18), using (19) and (20).

The learning rate γ_j should decrease to 0 as $j \to \infty$.

It is possible to update the quantities that are used by back-propagation efficiently as extra observations are added.

This is very desirable if the training set keeps growing over time.

In practice, people often use **stochastic gradient descent**, which deliberately introduces randomness. For example, the order in which parameters are updated may be determined randomly at each step.

11.2. Some Issues with Training Neural Networks

In general, the objective function for a neural net is not convex. Therefore, there may be multiple minima.

It is very important to start the algorithm from different places, and/or use methods such as simulated annealing, particle swarm, or genetic algorithms that are designed to handle non-convex functions.

Because neural nets have many parameters, overfitting is a problem. Deep neural nets can have millions of parameters!

One solution is to stop before getting to the overall optimum. This works best if the starting parameter values are small, so that the model is nearly linear.

ESL claims that a better approach is regularization, called **weight decay**. I don't know whether this is still recommended.

Instead of minimizing $R(\boldsymbol{\theta})$, we minimize

$$R(\boldsymbol{\theta}) + \lambda\left(\sum_{k,m}\beta_{km}^2 + \sum_{m,\ell}\alpha_{m\ell}^2\right). \tag{23}$$

This simply adds terms $2\lambda\beta_{km}$ and $2\lambda\alpha_{m\ell}$ to the derivatives (16) and (16), respectively, and these carry through to other equations.

The tuning parameter λ is normally chosen by cross-validation.

Instead of the penalty in (23), we could use the **weight elimination penalty**, which tends to shrink smaller weights more:

$$\lambda\left(\sum_{k,m}\frac{\beta_{km}^2}{1+\beta_{km}^2}+\sum_{m,\ell}\frac{\alpha_{m\ell}^2}{1+\alpha_{m\ell}^2}\right). \tag{24}$$

Figure 11.4 shows a classification example with and without weight decay (regularization). There are 10 hidden layers.

Without weight decay, there is severe overfitting, and performance on the test dataset is much worse than performance on the training dataset.

As with many machine learning methods, it makes sense to standardize the data to have mean 0 and variance 1 before beginning.

This is important for regularization and makes it easier to choose sensible starting values. ESL suggests that starting values should be $\mathrm{U}(-0.7, 0.7)$.

Question: What do we do if the sample size increases during the training process? With additional data, the entire sample will no longer be standardized.

But re-standardizing every time we get one or more additional observations would be expensive, would change estimates from the original sample, and would screw up procedures that update the estimates cheaply as additional data arrive.

ESL say that it is better to have too many hidden units than too few, since weights can always be shrunk by regularization. They suggest starting with 5 to 100 hidden units. Use higher numbers with larger training samples and more inputs.

Number of hidden layers varies with the problem. Choosing it requires experimentation and experience. The deep learning revolution has led to models with many more layers (50+ in some cases) than before.

Use the average predictions over a collection of (good) estimated networks. Because the NN model is nonlinear, this is *not* the same as averaging the weights over several sets of estimates.

Average the predictions over networks estimated using a number of bootstrap samples (bagging).

The zip code example in Section 11.7 is interesting.

Neural networks and projection pursuit take nonlinear functions of linear combinations of inputs.

Both can work well for prediction when the data contain quite a lot of information (high signal to noise ratio or very large sample size).

They are both hard to interpret, because each input can enter in many places, nonlinearly.

Because neural nets are smooth functions of real-valued parameters, it is natural to use Bayesian methods.

MCMC solves the problem of multiple local minima, and automatically provides averaging via posterior means. Prior information replaces regularization.

See Section 11.9 for a brief discussion of Bayesian neural nets and comparison with boosted trees, random forests, boosted neural nets, and bagged neural nets.