Variable Projection Applied to Neural Networks

Abraham Engle
Department of Mathematics
University of Washington
Seattle, WA 98195
aengle2@math.washington.edu

Christian Rudnick
Department of Mathematics
University of Washington
Seattle, WA 98195
rudnickc@math.washington.edu

Jair Taylor
Department of Mathematics
University of Washington
Seattle, WA 98195
jptaylor@math.washington.edu

Abstract

Neural networks are smooth, nonlinear, highly non-convex models with a large number of parameters. The gradient of the objective function with respect to the parameters may be computed using the backpropogation algorithm, and so gradient descent is traditionally used to train the weights in the network. In this work we investigate an alternative method due to Golub and Pereyra known as variable projection, in which the weights on the last layer of the network are considered to be functions of the inner layers, optimized at each time step, and gradient descent is performed only on the inner layers. Our numerical experiments indicate that variable projection is competitive when the size of the last layer is relatively large.

1 Introduction

Throughout the quarter, we considered predictive models of the form $y(\boldsymbol{x}, \boldsymbol{w}) = g\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$, where ϕ_j were potentially non-linear functions of the covariates $\boldsymbol{x}$ corresponding to response $\boldsymbol{y}$ and parameters $\boldsymbol{w}$ to be learned from an optimization method. A first example is the generalized linear model with canonical links, which led to robust estimates of the paraemters $\boldsymbol{w}$ of interest. A second example came from modifying the likelihood-based objective function from

logistic regression, which led to support vector machines with potentially non-linear kernels. We focused on another family of predictive models in the form given above, where the idea is to let the functions ϕ_j themselves on parameters to be learned. Neural Networks suggest the form of the ϕ_j to be $\phi_j(x) = g\left(\boldsymbol{\theta}^T \boldsymbol{x}\right)$, mimicking the framework above in a recursive manner, potentially multiple times.

2 Neural Networks

2.1 Model Description

Suppose we have a training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, with k features excluding intercept, i.e., $x_i \in \mathbb{R}^k$. The typical architecture of the feed-forward neural network is L "layers" with s_l nodes per layer, $l = 1, 2, \ldots, L$ excluding a count for the intercept. Training the the neural network for for a single sample (x_i, y_i) is visualized in the following schematic:

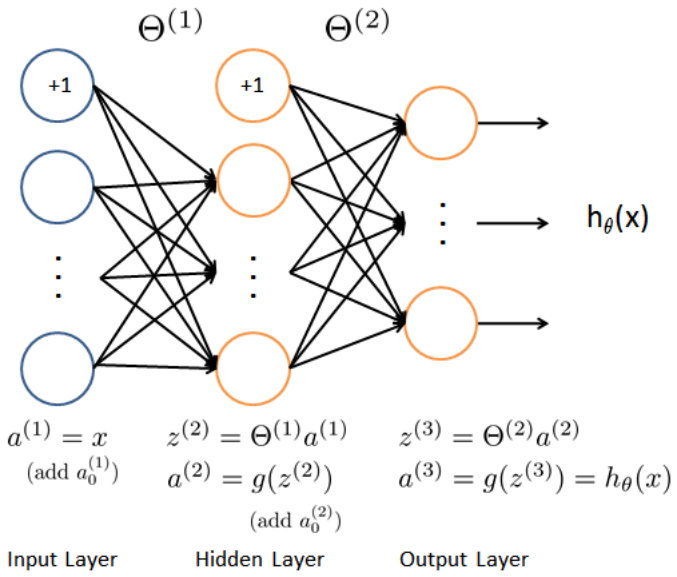

Figure 1: Neural Network Schematic from Andrew Ng Coursera Machine Learning

Each node (often called neuron) in the input layer above is just the components of the feature vector $x_i \in \mathbb{R}^k$. Subsequent layers are called hidden layers and the final layer is called the output layer, representing the fitted value(s) $\widehat{y}_i$. Nodes in the hidden layers are formed by applying an activation function $g(z)$–typically the sigmoid function as above but in principle any nice enough function mapping $\mathbb{R}$ to $(0, 1)$–to linear combinations of the previous nodes. An intercept term is artificially included into each layer of the network, and since each node has its own set of parameters, each corresponding layer therefore has a collection of parameters that we tabulate as the rows of a

parameter matrix Θ to be learned by an optimization routine. Sometimes a different final activation function used for the output layer in the network.

Fixing a notation, we let layer l in the neural network above be represented by a vector of values (the entries in the nodes) $a_k^{(l)}$, for $k = 1, 2, \ldots, s_l$ excluding a count for the intercept. Concretely, we have

$$x = a^{(1)} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_k \end{bmatrix}, \quad a^{(2)} = \begin{bmatrix} 1 \\ a_1^{(2)} \\ \vdots \\ a_{s_2}^{(2)} \end{bmatrix}, \quad \ldots, \quad a^{(L-1)} = \begin{bmatrix} 1 \\ a_1^{(L-1)} \\ \vdots \\ a_{s_{L-1}}^{(L-1)} \end{bmatrix}, \quad a^{(L)} = \widehat{y}.$$

Transitioning between layers of the neural network is given by the following pair of equations. Let

$$\begin{aligned} z^{(l+1)} &= \Theta^{(l)} a^{(l)} \\ a^{(l+1)} &= g(z^{(l+1)}), \end{aligned}$$

where $l = 1, 2, \ldots, L-1$ and $g(z^{(l+1)})$ denotes element-wise application of the activation. The quantities $\Theta^{(l)}$, representing the transition from layer l to layer $l+1$ are matrices of size $s_{l+1} \times s_l + 1$, for $l = 1, 2, \ldots, L-1$, assuming we use the same activation at the final layer. The framework presented here is often called the feed-forward neural network architecture, since fitted values $\widehat{y}_i$ are obtained by "feeding" the inputs through the network.

2.2 Objective Function

As is commonly done, we will use the ℓ^2 distance as a measure of error, and sum over all the n observations in the sample. The loss function to minimize is thus

$$E = \frac{1}{2} \sum_{i=1}^{n} \left\| \hat{y}_i(\Theta, x_i) - y_i \right\|_{\ell^2}^2$$

Here, $y_i \in \mathbf{R}^{m \times 1}$ is a response vector, whereas the vector $\hat{y}_i$ of the same dimension is the response predicted by the neural net. By expanding the squared ℓ^2-norm this can be rewritten as

$$E = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} \left| \hat{y}_{i,j} - y_{i,j} \right|^2.$$

To obtain the objective and gradient for a single hidden layer, we have

$$x_i \in \mathbf{R}^{s_0 \times 1}$$

and following earlier notation, we have

$$z_i = \Theta^{(0)} x_i + \Theta_0^{(0)} \in \mathbf{R}^{s_1 \times 1}, \qquad \Theta^{(0)} \in \mathbf{R}^{s_1 \times s_0}, \quad \Theta_0^{(0)} \in \mathbf{R}^{s_1 \times 1},$$

where we have separated out the intercept weights to ease notation for the differentiation to come. Applying the activation function g gives the values of the nodes in the hidden layer as

$$a_i = g(z_i) \in \mathbf{R}^{s_1 \times 1}.$$

Again we have suppressed superscripts to emphasize a single hidden layer. Finally, the fitted values are

$$\hat{y}_i = \Theta^{(1)} a_i + \Theta_0^{(1)} \in \mathbf{R}^{m \times 1}, \qquad \Theta^{(1)} \in \mathbf{R}^{m \times s_1}, \Theta_0^{(1)} \in \mathbf{R}^{m \times 1},$$

where we omit a final activation output for the rest of the paper. This gives

$$\hat{y}_i = \Theta^{(1)} a_i + \Theta_0^{(1)} = \Theta^{(1)} g\left(\Theta^0 x_i + \Theta_0^{(0)}\right) + \Theta_0^{(1)},$$

with corresponding objective

$$\begin{aligned} E &= E\left(\Theta^{(0)}, \Theta_0^{(0)}, \Theta^{(1)}, \Theta_0^{(1)}\right) \\ &= \frac{1}{2} \sum_{i=1}^{n} \|\hat{y}_i - y_i\|_{\ell^2}^2 \\ &= \frac{1}{2} \sum_{i=1}^{n} \left\| y_i - \left(\Theta^{(1)} a_i + \Theta_0^{(1)}\right) \right\|_{\ell^2}^2 \end{aligned}$$

2.3 Discussion of the Objective

We minimize the loss function $E(\Theta) = \frac{1}{2} \sum_{i=1}^{n} \left\| y_i - \widehat{y}_i(\Theta, x_i) \right\|_2^2$, where the hats indicate the fed-forward fits. This loss function unfortunately possesses some complicating properties. In general it is not convex and can and often does have several local minima. Moreover, the literature and experimental results indicate sensitivity to starting conditions that make training the neural network several times with different initial conditions a necessity. Reaching saddle points is another problem that can be encountered, so second derivative tests for local minima are often required.

Scaling the features is often employed prior to training the network to help with the optimization routine of choice, and the most basic of such is gradient descent, which minimizes the objective based on taking steps from the current iterate in the direction of the negative gradient. The step-size is a delicate matter, since taking very large steps in this direction need not necessarily decrease the objective for complicated enough surfaces. A line search that examines step lengths can be employed, and often second order methods of descent like quasi-Newton (approximations to the Hessian) are used in favor of gradient descent.

3 Optimizing the Objective

We present two methods of optimizing the objective function above.

3.1 Backpropagation of errors

The traditional approach is *backpropagation algorithm* which is simply gradient descent, i.e. one performs the iteration

$$E\left(\Theta^{(k+1)}\right) = E\left(\Theta^{(k)}\right) - \frac{1}{\eta}\nabla E\left(\Theta^{(k)}\right) \tag{1}$$

for a suitably chosen learning rate(step size) η. Here, Θ denotes the collection of all the parameters; in the case of a single hidden layer as above, we to compute the gradient of E given above. Observe from the definition of the forward equations that the partial derivatives can be computed by repeated use of the chain rule. To get a first feeling on how the process works, we will start with deriving the backpropagation for a single output and a single hidden layer. The case of multiple outputs is similarly handled by the way the loss function splits additively. With this simplification, the norm in the error function above is now just the difference of real numbers. Using vector notation, we have

$$\frac{\partial E}{\partial \Theta^{(1)}} = (\hat{y} - y)\, a.$$

Similarly,

$$\frac{\partial E}{\partial \Theta_0^{(1)}} = (\hat{y} - y)\, \frac{\partial}{\partial \Theta_0^{(1)}}\left(\Theta^{(1)} a + \Theta_0^{(1)}\right) = \hat{y} - y.$$

Next, we compute the partial with respect to $\Theta_{ij}^{(0)}$: Note that

$$\frac{\partial E}{\partial \Theta_{ij}^{(0)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial \Theta_{ij}^{(0)}}.$$

It is straightforward to see that

$$\frac{\partial E}{\partial \hat{y}} = \hat{y} - y, \qquad \frac{\partial \hat{y}}{\partial a} = \Theta^{(1)}, \qquad \frac{\partial z}{\partial \Theta_{ij}^{(0)}} = x_j e_i$$

where e_i is the ith unit vector in $\mathbf{R}^{s_1}$. For the third term,

$$\frac{\partial}{\partial z_i}\sigma(z) = \frac{\partial}{\partial z_i}\sigma(z_j) = \sigma'(z_j)\delta_{\{i=j\}},$$

so

$$\frac{\partial a}{\partial z} = \begin{pmatrix} \sigma'(z_1) & & \\ & \ddots & \\ & & \sigma'(z_{s_1}) \end{pmatrix} = \mathrm{Diag}(\sigma'(z)).$$

Thus

$$\frac{\partial z}{\partial \Theta_{ij}^{(0)}} = (\hat{y} - y)\Theta^{(1)}\mathrm{Diag}(\sigma'(z))x_j e_i = (\hat{y} - y)\Theta^{(1)}\sigma'(z_i)x_j e_i,$$

and in matrix notation,

$$\frac{\partial E}{\partial \Theta^{(0)}} = (\hat{y} - y)\left(\Theta^{(1)^T} \otimes \sigma'(z)\right)x^T.$$

Here $\otimes$ indicates pointwise multiplication of the corresponding matrices or vectors. Finally we can similarly derive that

$$\frac{\partial E}{\partial \Theta_{i0}^{(0)}} = \frac{\partial f}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial \Theta_{i0}^{(0)}} = (\hat{y} - y)\,\Theta^{(1)}\mathrm{Diag}(\sigma'(z))e_i,$$

or using concise notation,

$$\frac{\partial E}{\partial \Theta_{0}^{(0)}} = (\hat{y} - y)\,\Theta^{(1)}\mathrm{Diag}(\sigma'(z))\mathbf{1} = (\hat{y} - y)\left(\Theta^{(1)^T} \otimes \sigma'(z)\right)\mathbf{1}^T.$$

Next, we'll consider a neural net with an arbitrary number of layers L. As before, $a^{(0)} := x \in \mathbf{R}^{s_0 \times 1}$, but now we recursively compute for $\ell = 1, 2, \ldots L$,

$$\begin{aligned} z^{(\ell)} &= \Theta^{(\ell-1)}a^{(\ell-1)} \in \mathbf{R}^{s_\ell \times 1} + \Theta_0^{(\ell-1)}, \qquad \Theta^{(\ell-1)} \in \mathbf{R}^{s_\ell \times s_{\ell-1}}, \Theta_0^{(\ell-1)} \in \mathbf{R}^{s_\ell \times 1} \\ a^{(\ell)} &= \sigma\left(z^{(\ell)}\right) \in \mathbf{R}^{s_\ell \times 1} \end{aligned}$$

which corresponds to the propagation of the signal through the layers of the neural net. Finally, we obtain the output by computing

$$\hat{y} = \Theta^{(L)} a^{(L)} \in \mathbf{R} + \Theta_0^{(\ell)}, \qquad \Theta^{(L)} \in \mathbf{R}^{1 \times s_L}, \Theta_0^{(\ell)} \in \mathbf{R}.$$

The objective function is now a function of $2(L+1)$ matrices containing parameters,

$$E\left(\Theta^{(0)}, \Theta_0^{(0)}, \Theta^{(1)}, \Theta_0^{(1)}, \ldots, \Theta^{(L)}, \Theta_0^{(L)}\right) = \frac{1}{2}\left(\hat{y} - y\right)^2$$

To use gradient descent, just as before, we use the chain rule:

$$\frac{\partial E}{\partial \Theta_{ij}^{(L)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \Theta_{ij}^{(L)}} = (\hat{y} - y) a^{(L)}$$

and more generally, for $\ell \leq L - 1$,

$$\begin{aligned}
\frac{\partial E}{\partial \Theta_{ij}^{(\ell)}} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\mathrm{d}a^{(L)}} \left(\prod_{k=\ell+1}^{L} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial a^{(k-1)}} \right) \frac{\partial a^{(\ell)}}{\partial z^{(\ell+1)}} \frac{\partial z^{(\ell+1)}}{\partial \Theta_{ij}^{(\ell)}} \\
&= (\hat{y} - y) \Theta^{(L)} \left(\prod_{k=\ell+1}^{L} \mathrm{Diag}(\sigma'(z_1^{(k)}), \ldots, \sigma'(z_{s_k}^{(k)})) \Theta^{(k-1)} \right) \\
&\quad \cdot \mathrm{Diag}(\sigma'(z_1^{(\ell)}), \ldots, \sigma'(z_{s_\ell}^{(\ell)})) a_j^{(\ell)} e_i \\
&= (\hat{y} - y) \left(\prod_{k=\ell}^{L} \Theta^{(k)} \mathrm{Diag}(\sigma'(z_1^{(k)}), \ldots, \sigma'(z_{s_k}^{(k)})) \right) a_j^{(\ell)} e_i \\
&= (\hat{y} - y) \left(\prod_{k=\ell}^{L} \Theta^{(k)} \otimes \left(\sigma'(z^{(k)}) \mathbf{1}^T \right) \right) a_j^{(\ell)} e_i
\end{aligned}$$

In concise notation, this equals

$$\frac{\partial E}{\partial \Theta^{(\ell)}} = (\hat{y} - y) \left(\prod_{k=\ell}^{L} \Theta^{(k)} \otimes \left(\sigma'(z^{(k)}) \mathbf{1}^T \right) \right) a^{(\ell)} \mathbf{1}^T.$$

For the intercept matrix,

$$\frac{\partial E}{\partial \Theta_{i0}^{(\ell)}} = (\hat{y} - y) \left(\prod_{k=\ell}^{L} \Theta^{(k)} \otimes \left(\sigma'(z^{(k)}) \mathbf{1}^T \right) \right) e_i,$$

or using concise notation,

$$\frac{\partial E}{\partial \Theta_0^{(\ell)}} = (\hat{y} - y) \left(\prod_{k=\ell}^{L} \Theta^{(k)} \otimes \left(\sigma'(z^{(k)}) \mathbf{1}^T \right) \right) \mathbf{1}^T.$$

3.2 Variable projection

An alternative method[3, 4] to optimize the loss function is the variable projection method. It operates on the assumption that the residuals $r_i = \hat{y}_i - y_i$ are of the form

$$r_i = y_i - \sum_{j=1}^{d} b_j \phi_j(\beta, x_i) = y_i - \Phi(\beta) b$$

where β and the b_j denote model parameters, and that the loss function is of the form

$$E = \frac{1}{2} \sum_{i=1}^{n} \|r_i\|_{\ell^2}^2 = \frac{1}{2} \sum_{i=1}^{n} \left\| y_i - \Phi(\beta) b \right\|_{\ell^2}^2 = \frac{1}{2} \left\| y - \Phi(\beta) b \right\|_{\ell^2}^2.$$

This is basically the loss function of OLS regression and one can utilize this fact to simplify the estimation of the parameters. Assume that you have estimates for β, so that $\Phi(\beta)$ is known. Then one can find the vector b by simply performing OLS regression, that is,

$$b = \Phi(\beta)^{-} y \tag{2}$$

where Φ^{-} denotes a symmetric generalized inverse of the matrix Φ which satisfies that

$$\Phi \Phi^{-} \Phi = \Phi, \qquad \left(\Phi \Phi^{-} \right)^T = \Phi \Phi^{-}.$$

This generalized inverse is known to provide a minimum norm solution of the least squares problem. Now, unfortunately, we do not know β, but we do know the relationship between β and b and therefore we can express the loss function as being independent of b,

$$\begin{aligned} E &= \frac{1}{2} \left\| y - \Phi(\beta) b \right\|_{\ell^2}^2 \\ &= \frac{1}{2} \left\| y - \Phi(\beta) \Phi^{-}(\beta) y \right\|_{\ell^2}^2 \\ &= \frac{1}{2} \left\| \left(I - \Phi(\beta) \Phi^{-}(\beta) \right) y \right\|_{\ell^2}^2. \end{aligned}$$

One can now proceed to minimize this loss function to obtain an estimate for β, and the proceed to get a derived estimate for b by using equation 2.

Before we consider implementation details of this algorithm, note that the loss function of a neural net as discussed above can be expressed as

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{n}\left\| y_i - \left(\Theta^{(L-1)}a_i + \Theta_0^{(L-1)}\right)\right\|_{\ell^2}^2 \\ &= \frac{1}{2}\sum_{i=1}^{n}\left\| y_i - \sum_{j=1}^{s_{L-1}}\tilde{\Theta}_j^{(L-1)}\tilde{a}_{i,j}\right\|_{\ell^2}^2 \end{aligned}$$

where $\tilde{a}_i = (a_i, 1)^T$, and $\tilde{\Theta}^{(L-1)} = \left(\Theta^{(L-1)}, \Theta_0^{(L-1)}\right)^T$. Therefore, the variable projection method can be applied to neural nets where $b = \tilde{\Theta}^{(L-1)}$ is the vector containing the weights for the last layer.The minimization of this new functional can be done, once again, using gradient descent. To ease the notation somewhat, we will denote by

$$P_\Phi^\perp = I - \Phi(\beta)\Phi(\beta)^-$$

the variable projection operator (note that the notation is chosen to reflect that it is the projector on the orthogonal complement of the column space of $\Phi(\beta)$).

To minimize

$$E = \frac{1}{2}\left\|\left(I - \Phi(\beta)\Phi^-(\beta)\right)y\right\|_{\ell^2}^2 = \frac{1}{2}\left\|P_\Phi^\perp y\right\|_{\ell^2}^2$$

we will once agian use the gradient which can be computed using the chain rule,

$$\frac{\partial E}{\partial \beta_i} = P_\Phi^\perp y \frac{\partial}{\partial \beta_i} P_\Phi^\perp y,$$

so if we denote by $J(\beta) = P_\Phi^\perp y \frac{\partial}{\partial \beta_i}$the Jacobian of the variable projection operator, then

$$\nabla E = J^T(\beta) P_\Phi^\perp y.$$

Now one can run the gradient descent algorithm as indicated in equation **??** above[1]. The Jacobian in turn can be computed by the following formula

$$J_{\cdot j} = -\left[\left(P_\Phi^\perp \frac{\partial \Phi}{\partial \beta_j}\Phi^-\right) + \left(P_\Phi^\perp \frac{\partial \Phi}{\partial \beta_j}\Phi^-\right)^T\right] y.$$

[1]If one intends to use a second order method one can derive another formula for the Hessian which is given by

$$\nabla^2 E = J^T(\beta)J(\beta) + \sum_{i=1}^{n}(P_\Phi^\perp y)_i(\nabla^2(P_\Phi^\perp y)_i).$$

4 Results

Experiment 1. We wrote code in Python to implement a neural network and compare variable projection with the usual gradient descent algorithm using backpropogation. To compare the efficacy of the two algorithms, we used the following problem. Define the test function

$$f(x) = 3\cos(2x) + 4\cos(2.5x)$$

on the interval $0 \leq x \leq 10$. We then sampled $n = 300$ points x from the domain uniformly at random, using the first 150 as a training set and the second 150 as a test set. Given the training data $(x, f(y))$ for each sampled point, the algorithm must supply a continuous function $g(x)$ that approximates $f(x)$ well on the test set. That is, we aim to minimize the sum

$$\sum_x (f(x) - g(x))^2$$

where x ranges over the 150 sampled points in the test set.

The procedure is as follows. For each s in the set $\{10, 20, 30\}$ we create a neural net with a single hidden layer consisting of s nodes. At first we initialize the weights in the network randomly. We then use the two algorithms (described in detail below) to train the neural network, taking care that two are seeded with the same initial weights. We give each algorithm a time of two minutes to minimize the error, and then compare the results.

The procedure for each algorithm is as follows.

Gradient descent. Assuming all of the weights except for those on the last layer have been initialized randomly, we initialize the last layer weights by the least-squares minimizer as in the variable projection method. We perform backpropagation (gradient descent) on all parameters, implementing a line search method to find the optimal learning rate (step size) at each step.

Variable projection. We perform gradient descent on the projected (inner) problem. That is, we perform gradient descent on all parameters except those used in the final layer. After a gradient step (calculated with a line search), we calculate the solution to the outer problem via the least squares solution and continue the iteration

The results are summarized in the following table and figure

The two methods both begin to approximate the function extremely well on the test set, and it appears that variable projection affords moderate improvement over the alternating minimization algorithm.

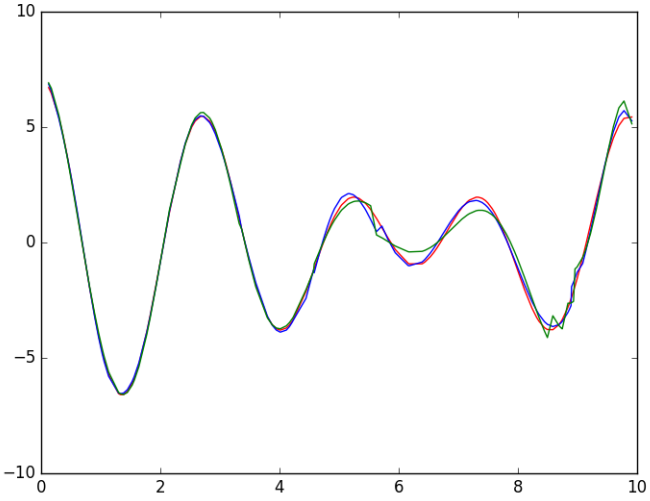

Figure 2: True Function (red), VP (blue), Gradient Descent (green)

Table 1: Training and test error for the univariate function.

Algorithm	# hidden nodes	10	20	30
Gradient descent	Avg. training error	1.03302249	0.0222784	9.44e-06
	Avg. test error	1.16233161	0.03373966	1.828e-05
Variable projection	Avg. training error	0.56302253	0.00619663	9.05e-06
	Avg. test error	0.76738657	0.01257624	1.826e-05

Experiment 2. For the second trial, we proceeded as before, but attempted to approximate a continuous function of two variables

$$f(s,t) = |st\cos(2.6(s+3))|$$

on the square $S = \{s,t|0 \leq s,t \leq 10\}$. This choice of $f(s,t)$ is intended to be complicated enough to be somewhat difficult to approximate accurately (rapidly changing and non-differentiable). We sampled $n = 1200$ points $x = (s,t)$, dividing it as before into 600 data points for training and 600 for testing.

The results are summarized in the following table. It appears that both methods take longer to learn

Table 2: Training and test error for the bivariate function.

	# hidden nodes	10	20	30	40	50
Gradient descent	Avg. training error	50.73	51.41	40.35	37.44	25.69
	Avg. test error	50.81	54.74	46.73	44.95	32.66
Variable projection	Avg. training error	40.96	30.87	35.16	30.9	19.2
	Avg. test error	45.13	35.36	41.97	39.3	24.95

the surface in terms of the MSE. Variable projection again appears to give moderate improvement over plain-vanilla gradient descent.

Experiment 3.

In a third simulation we tested the alternating minimization and variable projection methods on a real world dataset. [2] tried to predict the wine quality of white wine from certain physicochemical measurements such as the pH value or the amount of citric acid the wine contains. Similarly to what he had done before, we tested both algorithms on a neural net with one hidden layer of 10 to 40 nodes in increments of 10 nodes and recorded the average test and training errors. The results can be found in table 3.

Table 3: Training and test error for the wine dataset.

Algorithm	# hidden nodes	10	20	30	40
Gradient descent	Avg. training error	0.342	0.285	0.315	0.242
	Avg. test error	0.387	0.339	0.399	0.341
Variable projection	Avg. training error	0.347	0.345	0.315	0.286
	Avg. test error	0.393	0.419	0.847	0.342

As we can see, the gradient descent method consistently outperforms the variable projection method: The training and test errors for the variable projection method are consistently lower than than their counterparts for gradient descent. This stands in contrast to our findings when we approximated sinusoidal functions in one and two dimensions. This difference is at least partially due to the fact that there are more features in this simulation (13, whereas there were only one and two, respectively, in the prediction of the sinusoidal curve). Consequently, projecting out the parameters of the final layer should have less of an impact, thereby reducing the potential benefit of using variable projection.

5 Discussion and Extensions for Future Work

In class and in the textbook we discussed the result that a neural net with a single hidden layer can "uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided ... sufficiently large number of hidden units" [1]. Our results above suggest an experimental confirmation of this theoretical result.

Possible directions for future work include increasing the number of nodes in the input layer and determining a way to compare layer size and node size. The literature recommends that for larger input sizes, nodes in the hidden layer should roughly lie between the number of nodes in the output and input layers. We are also interested in comparing second order methods for minimizing the objective, and second order conditions should in practice be checked when the optimization algorithm

terminates. In practice networks are trained several times with different starting points to also ensure sufficient minimization of the objective.

A possible generalization is to relax the linearity assumption in the final layer. This is motivated by the general idea of non-linear variable projection, something we encountered early in the quarter. Supposing nonlinearity, we saw realized the Moreau envelope as an example of a type of nonlinear variable projection. The problem

$$y = y(x) = \operatorname{prox}_f(x)$$

is the solution of the projected minimization of the bivariate function $F(x, z) = \frac{1}{2}\|z - x\|_2^2 + f(z)$ for fixed x and any other infimal convolution $h = f\#g$ is another such example. We also encountered problems of the form

$$\widehat{a}(k) \in \arg\min_a \left\|A(k)a - y\right\|^2 + \lambda\|a\|_1 ,$$

in the final lecture of the course. This problem is solved using an interior point method and the "outer problem" of determining k uses a second order method like BFGS. Extensions in this direction include attempting to implement variable projection with a regularization parameter like this, which we found is already sometimes implemented in training networks. If we are interested in categorical prediction, the final layer of the optimization objective should also have an activation. Non-linear variable projection could be contrasted against the standard method of training for this case as well.

References

[1] C. M. Bishop, *Pattern recognition and machine learning*, Springer, New York, 2006.

[2] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, *Modeling wine preferences by data mining from physicochemical properties*, Decision Support Systems **47** (2009), no. 4, 547–553.

[3] G. H. Golub and V. Pereyra, *The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate*, SIAM Journal on Numerical Analysis **10** (1976), no. 3, 413–432.

[4] ———, *Separable nonlinear least squares: the variable projection method and its applications*, Inverse Problems **19** (2003), 1–26.