`netEvalFp`: Approximating $y = f(x)$ using a Neural Net and calculating its Jacobian

Alex Clymo

This note and repository stores some of what I am learning about implementing a neural net (NN) for function approximation in Matlab. It is mostly for my own consumption, to remind myself of these things once I forget them again, as there is a fair bit of technical stuff under the hood. The note presumes some basic knowledge of how neural nets work (but not much) and is more focused on explaining how they are implemented in Matlab, and how to use them for function approximation.

The repository also contains some functions I wrote for working with NN approximations. In particular, `dy_dx = netEvalFp(x,netParams)` returns the matrix of first derivatives (i.e. the Jacobian) of the approximated function at a point `x`, which is something you might need when, for example, evaluating the drift terms in an HJB. See Section 4 for more details of the code in this repository.

These notes build very heavily on the work of Alessandro Villa and Vytautas Valaitis, whose paper *A Machine Learning Projection Method for Macro-finance Models* (QE, 2024) is a fantastic reference for explaining the basics of neural nets and machine learning to a macroeconomist. My codes and knowledge build on the codes they made available at their Github repository here.

1 Our goal

The goal is to approximate a function $y = f(x)$ using a neural net. We will focus only on shallow neural nets with H nodes in the hidden layer. For our purposes, and to be consistent with how Matlab likes to handle the data for this kind of problem, we are looking at functions where the input x is a $R \times 1$ column vector and the output y is a $U \times 1$ column vector.[1] The NN simply approximates $f(x)$ using a certain functional form and parameters, to produce an approximation $f(x) \simeq NN(x, \hat{\phi})$ and predicted values $\hat{y} = NN(x, \hat{\phi})$, where $NN(x, \hat{\phi})$ is the approximating function evaluated at estimated parameter vector $\hat{\phi}$. See Appendix A for the explicit functional form used by the default Matlab neural net.

The parameters are trained to minimise an error function, typically the mean squared error. Stated like this, we see the problem is not so different from estimation or approximation ideas we are used to. One difference from, e.g., approximating a function using OLS is that the output y can be a vector, not just a scalar. To estimate the model (aka "train" the neural network) we simply give the code our x and y data and ask it to minimise the error, just like OLS would do. Suppose we have data with observations indexed by $i = 1, ..., Q$, where each

[1]Functions where the inputs and outputs are matrices or tensors can obviously be trivially handled by flattening them first.

datapoint is an observation of our input and output vectors x_i and y_i. For Matlab, we give the data to the training routine as matrices X and Y of size $R \times Q$ and $U \times Q$ respectively. This is called "supervised learning" because we are giving the model the Y data on which to train.

For many economic problems we would also like to know the first derivatives of our function after we approximate it. For our true function $y = f(x)$ this is the Jacobian matrix

$$J_f(x) = \frac{\partial f(x)}{\partial x^\top} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_U}{\partial x_1} & \cdots & \frac{\partial f_U}{\partial x_R} \end{bmatrix}$$

Once we have the estimated net, we can compute the approximation of the Jacobian as

$$J_{NN}(x) = \frac{\partial NN(x, \hat{\phi})}{\partial x^\top} = \begin{bmatrix} \frac{\partial NN_1(x,\hat{\phi})}{\partial x_1} & \cdots & \frac{\partial NN_1(x,\hat{\phi})}{\partial x_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial NN_U(x,\hat{\phi})}{\partial x_1} & \cdots & \frac{\partial NN_U(x,\hat{\phi})}{\partial x_R} \end{bmatrix}$$

We will code this up manually, as I do not believe Matlab provides a function to do it. See Section 4 for details of the code and Appendix A for the derivation of the Jacobian.

2 Creating a neural network in Matlab

- `feedforwardnet` sets up a generic shallow (i.e. one hidden layer) neural net. There are functions like `fitnet` which set up versions with specific purposes, which just tweak some of the settings. These two are similar enough you can choose either. To set up a net with `H` neurons in the hidden layer use

  ```
  net = feedforwardnet(H)
  ```

  The number of neurons is a choice which is up to you, like choosing what order of polynomial to use to approximate a function. Adding up all the weights and biases, the number of parameters the NN has is $H \times (R + U + 1) + U$, which can obviously very many parameters if the dimension of the input or number of nodes is high. The higher H the richer the approximation, but the more the model will try to overfit. This is where validation comes in, which is one difference from standard approximation techniques.

- Once created and trained, the net acts like a function you can call. E.g. if you train the net to approximate a function $y = f(x)$, this is supervised learning where the net will now store an approximation of the function, and return to you predicted values. Calling

  ```
  yhat = net(x)
  ```

  returns the predicted values `yhat` given an input `x`. At the same time, the `net` object acts like a structure, and you can inspect and edit using the structure notation `net.xyz` for any field `xyz`.

- The default settings for the NN in Matlab are:
  - Hidden layer has a `tansig` activation function, and output layer a `purelin` activation function

  - Inputs and outputs are scaled using `mapminmax`, which means they are scaled relative to their max and min values during processing.

3 Training the network

Once the neural network architecture has been defined and the input-output data prepared, training the network in MATLAB is performed using the `train` function. This adjusts the weights and biases to minimize the error between predicted and target outputs.

3.1 Basic Training Call

The network is trained via:

```
[net, tr] = train(net, x, y);
```

Here:

- `net` is the network object (e.g., created via `feedforwardnet(H)`).
- `x` is the input matrix of size $R \times S$, where S is the number of training examples.
- `y` is the matrix of target outputs of size $U \times S$.
- The returned `net` contains the trained parameters.
- The returned structure `tr` contains training diagnostics, including the indices used for training, validation, and testing, as well as training performance over iterations.

Each time the `train` function is called, the code starts at the currently stored parameters in `net` and updates them to get closer to the new data. This means that past training is always stored and built upon as long as you call and save to the same `net` object: we do not start again from scratch each time you call train.

3.2 Training, Validation, and Test Sets

MATLAB automatically divides the data into three sets:

- **Training set** – used to fit the weights and biases.
- **Validation set** – used to monitor generalization error during training. Training stops early if validation error worsens.
- **Test set** – used only for final performance evaluation and not accessed during training.

By default, this split is random via `'dividerand'`, which may yield different results across runs even with the same initial weights.

In economic modeling applications, such as equilibrium solvers or value function iteration, it is useful to eliminate randomness across training runs. For this purpose, MATLAB offers deterministic division via the `'divideint'` function, which divides data using interleaved indexing:

```
net.divideFcn = 'divideint';
```

This ensures the same data split is used on each run, promoting consistent behavior and easier convergence diagnostics. You can edit the fraction of observations allocation to each use by editing the parameters `net.divideParam.trainRatio`, `net.divideParam.valRatio`, and `net.divideParam.testRatio`.

3.3 Convergence Criteria in `train`

The `train()` function stops training when one of several criteria is met. For the default algorithm (`trainlm`), the key convergence conditions are:

- **Maximum number of epochs:** Training stops after a fixed number of iterations. Controlled by `net.trainParam.epochs` (default: 1000).
- **Minimum performance gradient:** Training stops if the gradient of the performance function falls below `net.trainParam.min_grad` (default: 10^{-7}).
- **Validation stop:** If the validation performance fails to improve for `net.trainParam.max_fail` consecutive checks (default: 6), training stops early.
- **Performance goal:** Training stops if the mean squared error drops below `net.trainParam.goal` (default: 0).
- **Maximum training time:** Training stops if it exceeds `net.trainParam.time` seconds (default: ∞).

Each of these parameters can be adjusted before calling `train`. For example:

```
net.trainParam.epochs = 500;  net.trainParam.min_grad = 1e-6;
```

Warning: the code will stop training after hitting the max number of epochs and will not loudly warn you that it stopped because it hit the max, and not because it converged. You can look at the training window output or inspect `tr.stop` to see why the training stopped. In the testing code I implemented this check and an ex-post calculation of the function fit (mean absolute error and R^2) to see how well the function converged

3.4 Additional notes on training

- Remember that the NN will try to fit the data the best it can. But if you do not allow enough neurons it will give a bad fit to the data. Check the fit after running training and consider increasing the number of nodes if the fit is bad. To avoid overfitting, the best practice would be (I think) to compute the fit only on the testing sample (which was not used in the training) when comparing hyper-parameters such as the number of nodes.
- The indices used for each subset are stored in `tr.trainInd`, `tr.valInd`, and `tr.testInd`.
- To manually specify the split, use `'divideind'` and set the index fields in `net.divideParam`.
- For full reproducibility across sessions, set the seed using `rng(seed)` at the top of your script.

- In many economic applications you might want to dampen the update of the approximation of a function. For example, we might re-train the net on a new simulation, and then use the net to produce a new simulation, and so on (think a Krusell Smith type application). To ensure stability, we might to not fully update the net each iteration.

  You can implement dampening of NN update in several ways. Note that calling `train` will by default try to exactly fit the data presented to it. To dampen, we need to essentially slow down or stop the `train` function before it does so. Here are some options:

  1. Simplest: Dampen the data being sent to train, then let train run as usual. E.g. don't send the new value functions, but a dampened update of them.
  2. Reduce number of epochs (`net.trainParam.epochs`). Epochs is number of iterations inside the train function, so fewer iterations might stop `train` from fully converging. However, it is typically very quick to converge quite close to the solution. Alternatively, just setting a low number of max epochs is also helpful for speeding up your code, as it might not be necessary to get full convergence during intermediate iterations.
  3. If using the default Levenberg–Marquardt algorithm for updating (`net.trainFcn = 'trainlm'`) increase initial dampening and slow how fast dampening is updated:

     `net.trainParam.mu = 1e-1` – Higher mu = more damping (default is 0.001)

     `net.trainParam.mu_dec = 0.1` – Decrease mu slowly

     `net.trainParam.mu_inc = 10` – Increase mu quickly on poor steps
  4. Switch to `net.trainFcn = 'traingd'` which has a fixed learning rate, and set a low learning rate `net.trainParam.lr = 0.001` and low number of max epochs. Apparently `traingda` is similar but with an adaptive learning rate.
  5. Manually blend the old and new NN parameters with dampening. We extract these parameters manually using the function `netExtractParams` we wrote, so could blend the old and new weights, biases, and scaling parameters.

4 Code details

This repository includes five MATLAB files: three functions and two example scripts. These provide tools for evaluating a trained feedforward neural network and demonstrating its use.

- `main_test1.m` and `main_test2.m`: Demonstration scripts showing how to:
  - Create and train a neural network using `feedforwardnet`.
  - Extract network parameters using `netExtractParams`.
  - Evaluate the output and Jacobian using the custom functions `netEvalF` and `netEvalFp`.

  These scripts serve as test beds for verifying the functionality of the custom code. The first script tests a function where y is a scalar output, and the second a function where y is a 3×1 vector output.

- `netExtractParams.m`: Extracts network parameters into a structured format for external use.

  Inputs: `net` — a trained feedforward neural network object.

  Outputs: `netParams` — a structure containing:
  - weights and biases for each layer,
  - input and output scaling parameters for mapminmax transformations.
  - Some basic details about the net structure

- `netEvalF.m`: Evaluates the output of the trained neural network at a batch of input points.

  Inputs:
  - `x` — matrix of input vectors, of size $R \times S$
  - `netParams` — structure of network parameters from `netExtractParams`

  Outputs: `y` — matrix of predicted outputs, of size $U \times S$

- `netEvalFp.m`: Evaluates the Jacobian of the network output with respect to the input, for a batch of input vectors.

  Inputs:
  - `x` — matrix of input vectors, of size $R \times S$
  - `netParams` — structure of network parameters

  Outputs: `J` — a 3D array of size $U \times R \times S$, where `J(:,:,s)` is the Jacobian at the s-th input point

Appendix

A Mathematical forms and derivations

A.1 Neural net functional form

Let $S_{\text{in}}(x)$ denote the elementwise affine transformation that rescales each component of the input vector $x \in \mathbb{R}^R$ from its original range to a target range, and let $S_{\text{out}}^{-1}(z)$ denote the inverse mapping applied to the output. Specifically:

$$\begin{aligned} S_{\text{in}}(x)_i &= (y_{\max,i}^{\text{in}} - y_{\min,i}^{\text{in}}) \cdot \frac{x_i - x_{\min,i}^{\text{in}}}{x_{\max,i}^{\text{in}} - x_{\min,i}^{\text{in}}} + y_{\min,i}^{\text{in}} \\ S_{\text{out}}^{-1}(z)_i &= \left(\frac{z_i - y_{\min,i}^{\text{out}}}{y_{\max,i}^{\text{out}} - y_{\min,i}^{\text{out}}} \right) \cdot (x_{\max,i}^{\text{out}} - x_{\min,i}^{\text{out}}) + x_{\min,i}^{\text{out}} \end{aligned}$$

Then the function form for a shallow neural network with H nodes in the hidden layer is:

$$\hat{y} = S_{\text{out}}^{-1} \left(W^{(2)} \cdot \tanh \left(W^{(1)} \cdot S_{\text{in}}(x) + b^{(1)} \right) + b^{(2)} \right)$$

where:

- $x \in \mathbb{R}^R$ is the raw input,
- $\hat{y} \in \mathbb{R}^U$ is the predicted output in the original (unscaled) space,
- $W^{(1)} \in \mathbb{R}^{H \times R}$, $b^{(1)} \in \mathbb{R}^H$, $W^{(2)} \in \mathbb{R}^{U \times H}$, $b^{(2)} \in \mathbb{R}^U$,
- $\tanh(\cdot)$ is applied elementwise.

A.2 Derivation of the Jacobian of the network output with respect to the input

We seek the Jacobian $J = \frac{\partial \hat{y}}{\partial x} \in \mathbb{R}^{U \times R}$. Define intermediate quantities:

$$\begin{aligned} z &= S_{\text{in}}(x) \in \mathbb{R}^R \\ a &= W^{(1)} z + b^{(1)} \in \mathbb{R}^H \\ h &= \tanh(a) \in \mathbb{R}^H \\ y^{\text{norm}} &= W^{(2)} h + b^{(2)} \in \mathbb{R}^U \\ \hat{y} &= S_{\text{out}}^{-1}(y^{\text{norm}}) \in \mathbb{R}^U \end{aligned}$$

Applying the chain rule:

$$J = \frac{\partial \hat{y}}{\partial x} = \frac{\partial \hat{y}}{\partial y^{\text{norm}}} \cdot \frac{\partial y^{\text{norm}}}{\partial h} \cdot \frac{\partial h}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial x}$$

We compute each term:

- $\frac{\partial \hat{y}}{\partial y^{\text{norm}}} \in \mathbb{R}^{U \times U}$ is diagonal with entries:

$$\left(\frac{\partial \hat{y}}{\partial y^{\text{norm}}}\right)_{ii} = \frac{x_{i,\text{out}}^{\max} - x_{i,\text{out}}^{\min}}{y_{i,\text{out}}^{\max} - y_{i,\text{out}}^{\min}}$$

- $\frac{\partial y^{\text{norm}}}{\partial h} = W^{(2)} \in \mathbb{R}^{U \times H}$
- $\frac{\partial h}{\partial a} \in \mathbb{R}^{H \times H}$ is diagonal with entries:

$$\left(\frac{\partial h}{\partial a}\right)_{ii} = 1 - \tanh^2(a_i)$$

- $\frac{\partial a}{\partial z} = W^{(1)} \in \mathbb{R}^{H \times R}$
- $\frac{\partial z}{\partial x} \in \mathbb{R}^{R \times R}$ is diagonal with entries:

$$\left(\frac{\partial z}{\partial x}\right)_{ii} = \frac{y_{i,\text{in}}^{\max} - y_{i,\text{in}}^{\min}}{x_{i,\text{in}}^{\max} - x_{i,\text{in}}^{\min}}$$

Combining all terms, the Jacobian becomes:

$$J = D_{\text{out}} \cdot W^{(2)} \cdot D_{\tanh} \cdot W^{(1)} \cdot D_{\text{in}}$$

where:

$$\begin{aligned} D_{\text{out}} &= \operatorname{diag}\left(\frac{x_{i,\text{out}}^{\max} - x_{i,\text{out}}^{\min}}{y_{i,\text{out}}^{\max} - y_{i,\text{out}}^{\min}}\right) \in \mathbb{R}^{U \times U} \\ D_{\tanh} &= \operatorname{diag}\left(1 - \tanh^2(a_i)\right) \in \mathbb{R}^{H \times H} \\ D_{\text{in}} &= \operatorname{diag}\left(\frac{y_{i,\text{in}}^{\max} - y_{i,\text{in}}^{\min}}{x_{i,\text{in}}^{\max} - x_{i,\text{in}}^{\min}}\right) \in \mathbb{R}^{R \times R} \end{aligned}$$