TensorFlow Implementations Part 1

By
Al Bernstein
6/22/2021
http://www.metricmath.com
al@metricmath.com

Introduction

This writeup goes into some basics of how TensorFlow works and an implementation in TensorFlow for fitting a polynomial. This is part 1 and covers forward calculations only. In Tensorflow, the data is represented by the tf.Tensor class and the neural layers are represented by the tf.Layer class where tf is the variable representing the python tensorflow module.

Tensors

A tensor in physics is a linear mathematical object whose properties are invariant with respect to coordinate transforms. A scalar is independent of coordinate changes and is a zero rank tensor. A vector is a single geometric object whose components change under coordinate transformations although its geometric properties remain the same. Figure 1 shows a vector in an xy coordinate system.

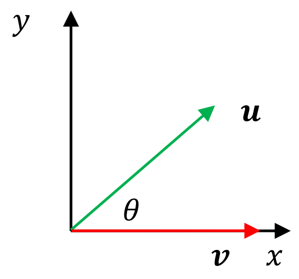

Figure 1

The vector's properties - magnitude $|\boldsymbol{v}|$and angle θ - are invariant under coordinate transforms and therefore the dot product is invariant as well – see equation (1). A vector is a first rank tensor.

$$\boldsymbol{v}\cdot\boldsymbol{x} = \boldsymbol{v}'\cdot\boldsymbol{x}' = |\boldsymbol{x}||\boldsymbol{v}|cos(\theta) = constant \Rightarrow$$

$$\boldsymbol{u}\cdot\boldsymbol{v} = \boldsymbol{u}'\cdot\boldsymbol{v}' = constant \tag{1}$$

The arbitrary vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ must transform in a way to keep the dot product constant. For a simple example, let A be a transformation from Cartesian to Polar coordinates.

$$A = \begin{bmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{bmatrix} \tag{2}$$

Define two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$.

$$\boldsymbol{u} = \frac{1}{\sqrt{2}}\hat{\boldsymbol{x}} + \frac{1}{\sqrt{2}}\hat{\boldsymbol{y}} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \tag{3}$$

$$\boldsymbol{v} = \hat{\boldsymbol{x}} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{4}$$

Set $\alpha = \frac{\pi}{3}$ in equation (2) $\Rightarrow$

$$A = \begin{bmatrix} 1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix}$$

$$\boldsymbol{u}' = \begin{bmatrix} \frac{1+\sqrt{3}}{2\sqrt{2}} \\ \frac{-\sqrt{3}+1}{2\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

$$\boldsymbol{v}' = \begin{bmatrix} \frac{1}{2} \\ \frac{-\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} 1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\boldsymbol{u} \cdot \boldsymbol{v} = \frac{1}{\sqrt{2}} = \boldsymbol{u}' \cdot \boldsymbol{v}' = \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}} = \frac{4}{4\sqrt{2}} = \frac{1}{\sqrt{2}} \tag{5}$$

Equation (5) works because this is a special case where A is an orthonormal matrix and $\boldsymbol{xy}$ is an orthonormal coordinate system - so

$$A^{-1} = A^T \Rightarrow (A^{-1})^T = A \tag{6}$$

For non-orthonormal coordinate systems, we would transform one of the vectors using $(A^{-1})^T$- see[1][2] for a more detailed explanation.

[1] General Coordinates

[2] Curvilinear Coordinates

Another example is the metric tensor G which is a 2^{nd} rank tensor that defines distance and is represented by a matrix. The distance is invariant under a coordinate transform as shown in equation (7).

$$\boldsymbol{x^T} G\boldsymbol{x} = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & \ddots & \vdots \\ g_{n1} & \cdots & g_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = g_{ij} x_i x_j = \boldsymbol{x^{T}}' G' \boldsymbol{x}' \equiv distance^2 \equiv \text{Invariant} \tag{7}$$

The vector $\boldsymbol{x}$ and tensor G components transform to keep the $distance^2$ invariant.

Tensors are mathematically represented by scalars, vectors, matrices, and multidimensional arrays. In machine learning, a tensor is a scalar, vector, or multidimensional array. Some examples are listed.

Scalar $\equiv \; [a] \equiv 0^{th}$ rank tensor

Vector $\equiv \boldsymbol{v} = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = v_i \equiv 1^{st}$ rank tensor

Metric $G = \begin{bmatrix} G_{11} & \cdots & G_{1n} \\ \vdots & \ddots & \vdots \\ G_{n1} & \cdots & G_{nn} \end{bmatrix} = G_{ij} \equiv 2^{nd}$ rank tensor

3^{rd} rank tensor $T = \begin{bmatrix} M_1 & \cdots & M_n \end{bmatrix} = T_{ijk}$ where M_n are 2^{nd} rank tensors

Higher order tensors continue using this same pattern. Because tensors have invariant properties under coordinate transforms, equations involving tensors have the same form in all coordinate systems. If a tensor equation is true in one coordinate system, then it is true in all coordinate systems. Equations (1) and (7) are examples of tensor equations.

Tensor Notation

Tensors can be represented in an index notation called tensor notation. Component versions of the above are shown below.

Vector $\equiv v_i$

$$\text{Inner product} \equiv u_i v_i = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \tag{8}$$

as defined in equation (1) is constant (invariant).

Repeated indices have an implied summation and is called Einstein's convention. So,

$$u_i v_i = \sum_i u_i v_i \tag{9}$$

$$\text{Outer product} \equiv u_i v_j = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1 v_1 & \cdots & u_1 v_n \\ \vdots & \ddots & \vdots \\ u_n v_1 & \cdots & u_n v_n \end{bmatrix} \tag{10}$$

where

the normal rules of matrix multiplication are used.
In this case there are no repeated indices, so there are no implied sums.

Matrix Multiplication of column vector $\equiv \boldsymbol{u} = u_i = M_{ij} v_j = M\boldsymbol{v} \Rightarrow$

$$\boldsymbol{u} = \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \tag{11}$$

Note: there is an implied sum over j.

Matrix Multiplication of row vector $\equiv \boldsymbol{u^T} = u_j = v_i M_{ij} = \boldsymbol{v^T} M \Rightarrow$

$$\boldsymbol{u^T} = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix} = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{bmatrix} \tag{12}$$

Note: there is an implied summation over i.

Tensors in Tensorflow

Tensors in machine learning are the mathematical structures that are used to represent tensors - scalars, vectors, matrices, and multidimensional arrays. For this discussion, tf represents the python tensorflow module and np represents the python numpy module.

```
import tensorflow as tf
import numpy as np
```

There are two ways to instantiate a variable in tensorflow 2.0.

1.) tf.constant$(value,\ dtype)$ → values cannot be changed
2.) tf.variable$(value,\ dtype)$ → values can be changed

where

value is a python list and gets turned into a numpy array
dtype is the numeric type – see documentation - Tensorflow dtypes

Example:

test =tf.constant($[1 \quad 2], \; tf.float32$)

Figure 2 shows the output of the variable test in a python console where TensorFlow's eager execution has been enabled so tensors are evaluated immediately.

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([1, 2])>
```

Figure 2

Note: arrays are defined in terms of nested lists in python, for example,

A vector $\equiv v = [v_1 \quad \cdots \quad v_n]$

An $m \times n$ matrix $\equiv M = [[M_{11} \quad \cdots \quad M_{1n}] \quad \cdots \quad [M_{m1} \quad \cdots \quad M_{mn}]] \rightarrow$

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is represented by the list $[[1 \quad 2 \quad 3] \quad [4 \quad 5 \quad 6]]$ and is implemented in numpy by the following statement:

np.array($[[1 \quad 2 \quad 3] \quad [4 \quad 5 \quad 6]]$)

The pattern continues for higher rank tensors.

Tensor Algebra in TensorFlow

The functions tf.matmul and tf.linalg.matmul perform matrix multiplication but need the input to be two matrices, and causes an error when trying to multiply a matrix by a vector. This is a less efficient way to perform algebra involving both vectors and matrices because vectors would have to be converted to matrices. An extremely flexible way to perform tensor algebra in TensorFlow is using the function tf.einsum which implements the Einstein summation index notation. If there are both vectors and matrices, tf.einsum can use them directly without conversion. Another advantage of tf.einsum is that it generalizes to expressions with tensors of rank greater than two.

The signature for tf.einsum is tf.einsum(equation, *inputs)

where

> *inputs ≡ a list of tensors – for example $\boldsymbol{u}, \boldsymbol{v}$
>
> equation ≡ a string representation of indices used in the expression. For example, the expression $u_i v_j$ would lead to the equation variable $'i, j'$ which represents the indices in the equation. Note: the comma separates tensors in an expression $\Rightarrow$ $'i'$ is used with u and $'j'$ is used with v. An arrow can be used in this string but is not always necessary $\Rightarrow$

$'i,j \rightarrow ij'$. In the following examples, the arrow is only necessary for the matrix transpose.

Define the following:

$\boldsymbol{u} = [3 \quad 4]$
$\boldsymbol{v} = [5 \quad 6]$

$A = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}$

$\boldsymbol{w} = [11 \quad 12]$

Outer Product

The outer product of $\boldsymbol{u}$ and $\boldsymbol{v}$ would be implemented as

$$\text{tf.einsum}('i,j', u, v) = u_i v_j = \boldsymbol{u^T v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} [5 \quad 6] = \begin{bmatrix} 15 & 18 \\ 20 & 24 \end{bmatrix} \tag{13}$$

The equation variable could also be written as $'i,j \rightarrow ij'$ which in this case is not necessary.

Inner Product

To implement an inner product of u and v, use $'i,i'$ or $'i,i \rightarrow '$ as the index equation as shown below.

$$\text{tf.einsum}('i,i', u, v) = u_i v_i = 3 \times 5 + 4 \times 6 = 15 + 24 = 39 \tag{14}$$

Note that the index equation $'i,i'$ uses the Einstein summation convention of summing over repeated indices.

Matrix Multiplication of a Column Vector

$$\text{tf.einsum}('ij,j', A, u) = A_{ij} u_j = A = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 53 \\ 67 \end{bmatrix} \tag{15}$$

Matrix Multiplication of a Row Vector

$$\text{tf.einsum}('i,ij', u, A) = u_i A_{ij} = \boldsymbol{u^T} A = [3 \quad 4] \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix} = [57 \quad 64] \tag{16}$$

Outer Product of Three Vectors

$$\text{tf.einsum}('i,j,k', u, v, w) = T_{ijk} = u_i v_j w_k = \left[\begin{bmatrix}165 & 180\\198 & 216\end{bmatrix} \quad \begin{bmatrix}220 & 240\\264 & 288\end{bmatrix}\right] \tag{17}$$

The indices are in sequential order. Here the indices associated with equation (17).

$$i,j,k \;\rightarrow \left[\begin{bmatrix}000 & 001\\010 & 011\end{bmatrix} \quad \begin{bmatrix}100 & 101\\110 & 111\end{bmatrix}\right] \tag{18}$$

where i, j, k are indexed 0,1

Matrix Transpose

This is a case where the arrow → is necessary.

$$\text{tf.einsum}('ij \;\rightarrow ji', A) \equiv \; A_{ij}^{T} = A_{ji} \tag{19}$$

Layers in TensorFlow

Layer is defined by tf.keras.layers.Layer and is the base class for all other layer classes in tensorflow – see Layer.

Layer classes inheriting from Layer can implement the following methods

1.) __init__(self) – input variables of class
2.) build(self, input_shape) – sets up weights and biases
3.) call(self, _inputs) - _inputs are a tensor of inputs

The call method performs the computation on the input tensor and it implicitly calls the build method. The call method is called when passing an input tensor to the layer as shown in equation (20).

$$\boldsymbol{y} = customLayer(\boldsymbol{x}) \tag{20}$$

where

$\boldsymbol{y}$ is the resultant tensor
$\boldsymbol{x}$ is the input tensor

Processing Weights and Biases

In TensorFlow, the activation functions operate on tensors and is the reason there is only one activation function per layer. Figure 3 shows the inputs fed to the activation functions using weights and biases in a very simple neural network.

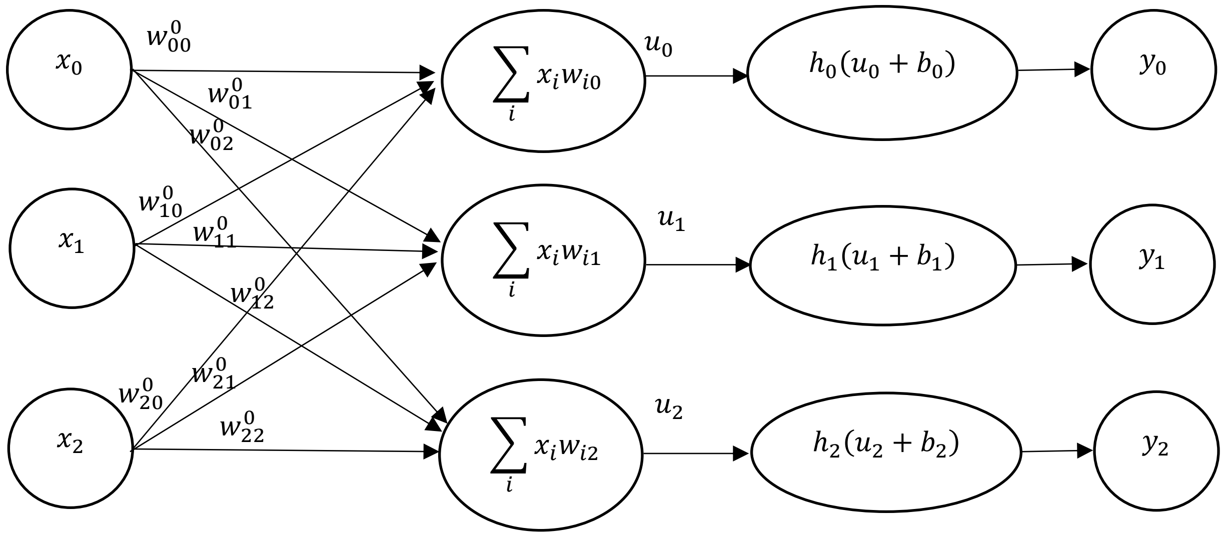

Figure 3

Note: the subscripts of the weights give the node numbers forming the connection. The superscript is the layer number, so $w_{from\ to}^{layer}$ is the format $\Rightarrow$ w_{01}^0 means 0^{th} layer from x_0 to h_1.

Figure 3 can be thought of in terms of vectors and matrices. Equation (21) shows the equation for u_0.

$$u_0 = \sum_i x_i w_{i0}^0 = \begin{bmatrix} w_{00} & w_{10} & w_{20} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{21}$$

Nodes u_1, and u_2 use the same pattern $\Rightarrow$

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = W\boldsymbol{x} \tag{22}$$

Adding the biases and feeding into the activation function $H \Rightarrow$

$$\boldsymbol{y} = H(W\boldsymbol{x} + \boldsymbol{b}) \tag{23}$$

where

$\boldsymbol{b}$ is the bias vector
$\boldsymbol{x}$ is the input vector
H is the activation function

Note: equation (23) implies that $h_0(x), h_1(x), h_2(x)$ are the same function $h(x)$ and $H(X)$ operates on the entire layer applying the activation function on the tensor X.

Equation (23) is what would be implemented in the call method described above to implement the neural network in Figure 3.

If x_0 ,⋯, x_n are vectors, then $\boldsymbol{x} \Rightarrow$

$$X = \begin{bmatrix} \boldsymbol{x_0^T} \\ \vdots \\ \boldsymbol{x_n^T} \end{bmatrix} = \begin{bmatrix} x_{00} & \cdots & x_{0n} \\ \vdots & \ddots & \\ x_{n0} & \cdots & x_{nn} \end{bmatrix} \tag{24}$$

and equation (23) $\Rightarrow$

$$Y = H(WX + \boldsymbol{b}) \tag{25}$$

Note: the bias $\boldsymbol{b}$ and is added to a matrix which is not mathematically allowed. In tensorflow, adding a scalar to a vector using the " $+$ " operator adds the scalar to each component of the vector. In equation (25), adding a bias vector to a matrix adds the b_i component to the i^{th} row of the matrix.

TensorFlow Implementation of Neural Fit

Figure 4 shows the neural net from Universal Approximation Theorem.

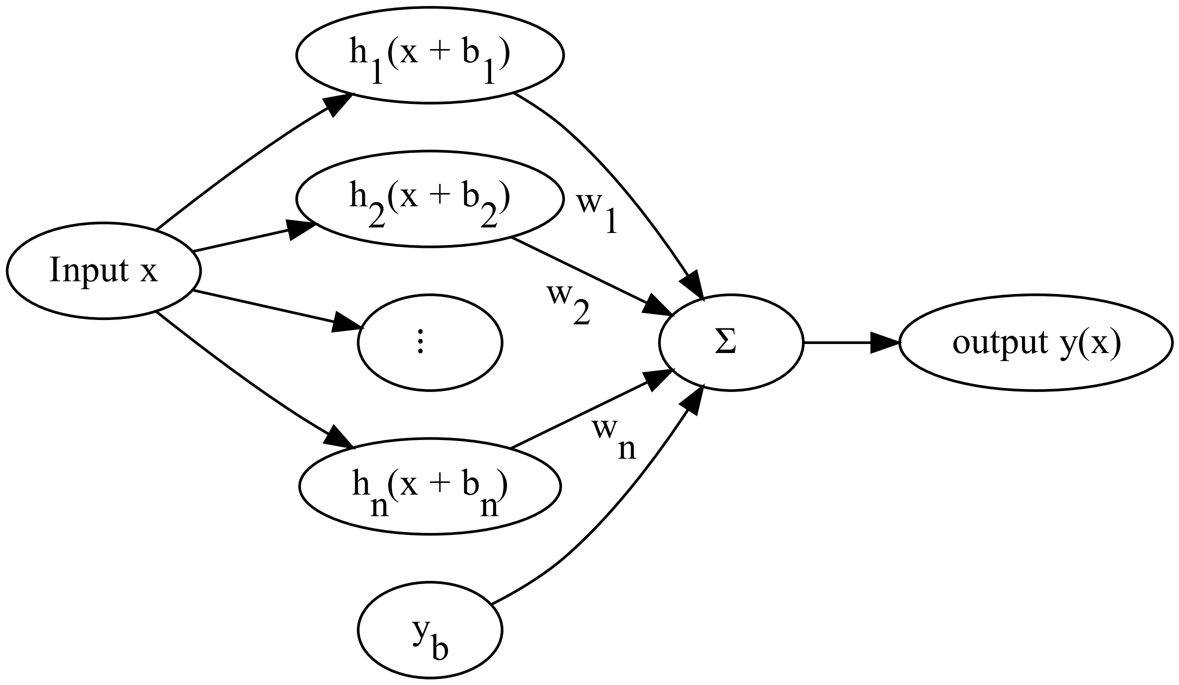

Figure 4

To implement Figure 4 in TensorFlow, we use three layers. The first is a fan-out layer that takes the input and stacks it so it can be fed into multiple neurons. The second is an ReLU layer that, adds the bias terms to the input and computes the ReLU function on the result. This implements equation (23) with $W = I$. The third is a summation layer that adds weighted outputs from multiple neurons and is equation (23) with $\boldsymbol{b} = 0$ and no activation function.

1.) Fan-out Layer

The Fan-out layer stacks the input $\boldsymbol{x}$, so it can be fed into multiple neurons as shown in Figure 4 and Equation (26).

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix}$$

$$X_1 = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \otimes \boldsymbol{x} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} [x_0 \quad \cdots \quad x_n] = \begin{bmatrix} x_0 & \cdots & x_n \\ \vdots & \ddots & \vdots \\ x_0 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{x^T} \\ \vdots \\ \boldsymbol{x^T} \end{bmatrix} \tag{26}$$

where

X_1 is the matrix output of the Fan-out layer
$\otimes$ is the outer product operator
$\boldsymbol{x^T}$ is the input vector as a row in $X_1 \Rightarrow$ each row is a replicated input vector.

2.) ReLU Layer

The ReLU Layer adds a bias term to each row of the input X_1 and feeds the result into the ReLU function. The weight matrix is implicitly the identity matrix and this layer implements equation (23) with $H(X) = ReLU(X)$.

3.) Summation Layer

The summation layer is the $\bigcirc\!\!\!\!\!\Sigma$ node in Figure 4 which performs the weighted sum of $ReLU(X + \boldsymbol{b})$ as shown in equation (27).

$$y_j = y_b + \sum_i w_i X_{1_{ij}} \tag{27}$$

where

w_i is the i^{th}component of the weight vector
$X_1 \equiv$ matrix output from the Fan-out layer
$y_j \equiv$ is the output row vector
$y_b \equiv$ is the system bias

Use tf.einsum to implement equation (27) $\Rightarrow y =$ tf.einsum$('i, ij', w, X_1)$

Figure 5 shows the result of fitting $^1/_2\,(5x^3 - 3x)$ with 20 neurons in the interval $[-1 \quad 1]$ implemented with tensorflow – equation (23). Note: the graph uses the interval $[-2 \quad 2]$ to show the behavior outside of the $[-1 \quad 1]$ interval and the mean square error is computed in the $[-1 \quad 1]$ interval only.

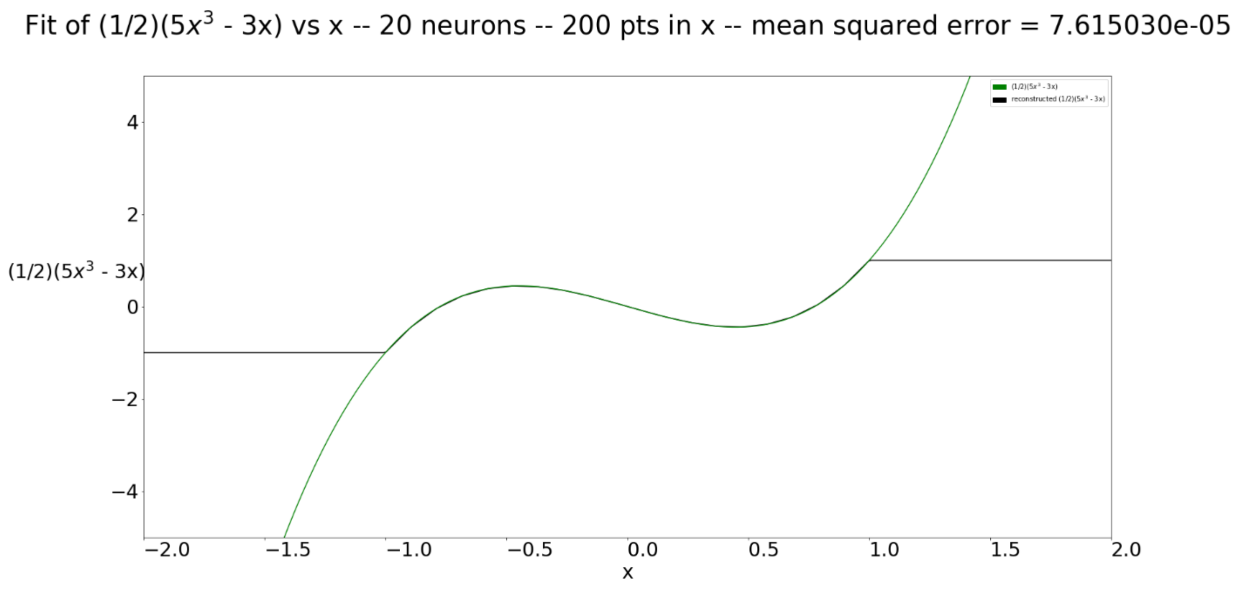
Fit of (1/2)(5x^3 - 3x) vs x -- 20 neurons -- 200 pts in x -- mean squared error = 7.615030e-05
(1/2)(5x^3 - 3x)
reconstructed (1/2)(5x^3 - 3x)
(1/2)(5x^3 - 3x)
4
2
0
−2
−4
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

Figure 5