Feedforward Artificial Neural Network Learning of Particle Movement Consistent With Conservation Laws

Lawson Fuller
University of California, San Diego
La Jolla, CA 92093
llfuller@ucsd.edu

Abstract

Animals can intuitively recognize and predict collisions (or other interactions) between external objects by observing the objects' states at different locations and times. Motivated by the idea of implementing this ability in machines, this project's goal is to explore whether artificial neural networks (ANNs) can learn this prediction and classification process by perfect observation of moving particle data. The feedforward networks in this work demonstrate the ability to learn three simple rules (but not a cubic function) and an ability to predict one particle's one dimensional position and momentum with negligible error. However, the feedforward networks which were tested fail to learn two-dimensional motion prediction. Another network learns to predict post-collision velocities of two particles from pre-collision velocities in two dimensions which is consistent with momentum and energy conservation with a large standard deviation. Attempts to train classification of collisions versus non-collisions from initial position and velocity data of two particles with nonzero radii do not succeed with the feedforward neural networks used in the project, which suggests that a different network architecture may be better suited to the classification task.

1 Introduction

Feedforward networks are directed acyclic sequences of node layers. Inputs given at one end are processed by the following layers in sequence, with the final layer producing the output. A densely connected layer is a layer in which each of its neurons receives inputs from all neurons in the preceding layer and outputs to all neurons in the following layer. Each densely connected layer has multiple nodes (or "neurons") which individually accept the sum of weighted inputs from all neurons in the preceding layer and store that sum in a single number z:

$$z_{l,j} = \Sigma_{i=1}^{n} w_{l,i,j} x_{l-1,i} + b_l.$$

The layer is labeled by l. Index i labels the output of the neuron i in layer $l-1$, index j labels the neuron in the index j of layer l, and b_l is some scalar bias. That sum is mapped to a new number using some designated function called an "activation function".

One example of an activation function is the sigmoid

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

Another example is the Leaky ReLU (leaky rectified linear unit) activation function:

$$LeakyReLU(z,\alpha) = \begin{cases} \alpha z & z < 0 \\ z & z \geq 0 \end{cases}$$

with ReLU being a special case where parameter $\alpha = 0$.

An optimization algorithm optimizes the weights w of the network so that the output of a given input attempts to minimize the cost function. Cost functions are the metric by which "inaccuracy" of the predicted output vs target output is measured. The target output is the desired output of a given input. By training the network on input vectors with their corresponding predicted outputs and targets, the neural network weights are optimized according to the optimization algorithm. The network is said to have "learned" from the data.

If trained too much on a particular set of inputs and targets, the network weights can over time become "overfitted" to the training data so that it does not predict well with any new data. This means that it is necessary to test the network predictions using additional "validation" inputs and their corresponding validation outputs which the network was not trained on. Normally there is also a test set which we predict on at the very end of the parameter and architecture choice process, but I will apply this step only to the NSKTPC network (acronym defined in Table 1).

Neural networks containing only linear activation functions will output only linear combinations of the input values. Therefore the network learns a linear function. If the activation functions are nonlinear, with popular choices such as the sigmoid or "ReLU" (rectified linear unit) functions, then the network can learn nonlinear mappings from the input to the output.

One simple example of interaction, and the focus of this project, is a collision between two massive particles in a vacuum. The collision is "elastic" if the energy and momentum of the total system remain unchanged by the collision. Although the kinematic equations describe a single particle's position, velocity and acceleration through time (with simple linear relationships in the case of zero acceleration), the relation between pre and post elastic collision velocities of two particles in two dimensions is more complex. We will first test on much simpler function learning and then explore the ability of a feedforward network to learn the physical relations.

It is often easiest to begin with calculations in the center of momentum frame and then transform coordinates to the frame of measurement (the "lab frame"). The center of momentum frame (also called the center of mass frame or COM) is the frame of reference in which the total system momentum is zero and the center of mass is at the origin. Center of momentum frame calculations can be generalized to any two-dimensional collisions through first rotating by an angle in the two-dimensional plane and then applying the center of momentum (or identically, mass) velocity to all particles in the system. That is, for a single particle

$$V_{Lab} = RV_{inCOM} + V_{cm},$$

where V_{cm} is velocity of the center of mass in the lab frame, R is the rotation matrix for rotation in the x-y plane, and V_{inCOM} is the velocity of the particle in the center of mass frame.

2 Methods

The feedforward networks are built in Python 3.7 using the Tensorflow API Keras and the scipy package.

2.1 Data Generation

A neural network needs to train on data in order to learn the appropriate weights. Two codes named SimulateCollisions.py and SimulateMisses.py generate the data in the laboratory reference frame for particles during some timespan. They then save the data into files that the network programs can load the samples from. Each time series of pair states is a sample, and many samples are generated and saved simultaneously.

The SimulateCollisions.py code first randomly generates point particle masses, x-axis positions, and x-axis velocities (aimed toward the origin) for both particles consistent with the COM frame, with the collision point at the origin. This is called the aligned COM frame. The positions and velocities are then rotated by a random angle ϕ, so that the particles are still in the COM frame, but unaligned with the new x-axis. Finally, a new random COM velocity is applied to both particles' velocities. This final set of velocities and positions over time is measured in the laboratory frame. All states (positions and momenta) for both particles for times before and after the collision are determined by

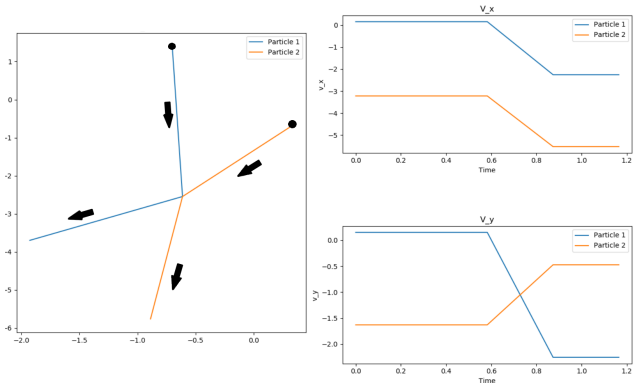


Figure 1: Example of a five timestep elastic collision generated by SimulateCollisions.py.

combining pre and post collision vectors for each quantity, joined at the moment of collision. These time-dependent state vectors and total system energy and momentum are generated twice and stored in training and validation files from which they can be extracted.

In SimulateMisses.py, the masses have a radius r. The positions and momenta are generated first in the unaligned COM frame, with velocities and positions generated as radial vectors with randomized magnitude and direction. It is inefficient to iterate the particles forward by small timesteps to check for collisions. In order to efficiently determine whether two particles with the generated initial conditions will collide, we can first determine the minimal distance between them using differentiation. The square of the distance between particle 1 and particle 2 at any time t is

$$d^2(t) = ((x_{1,i} + v_{1,x}t) - (x_{2,i} + v_{2,x}t))^2 + ((y_{1,i} + v_{1,y}t) - (y_{2,i} + v_{2,y}t))^2.$$

Next we calculate the time t_{min} at which $d^2(t)$ is minimized, which is

$$t_{min} = -\frac{(v_{1,x} - v_{2,x})(x_{1,i} - x_{2,i}) + (v_{1,y} - v_{2,y})(y_{1,i} - y_{2,i})}{(v_{1,x} - v_{2,x})^2 + (v_{1,y} - v_{2,y})^2}.$$

If $t_{min} \geq 0$ and $d(t_{min}) < 2r$, then a collision occurs at or after the initial time and the sample is classified as a collision. If this is not the case, then it is classified as a miss instead of a collision.

2.2 Learning

There are multiple learning codes, each serving a different purpose. Each learning code has an acronym listed in Table 1, and the codes' purposes, inputs, and outputs are listed in Table 2.

Net_Simple_Test.py (NST) tests the ability of fully linear and fully LeakyReLU networks to learn very simple output patterns. It learns are to produce output identical to the input, zeros for all vector elements regardless of input, an output which is an alternating pattern of elements, and cubic power values of the inputs. This code has only two training samples and two corresponding outputs, which the learning algorithm processes many times. As such, overfitting is a concern. I include results here only for the two LeakyReLU layer network.

NSKTPC predicts a single particle's one-dimensional state (position and velocity) at time $t + \Delta t$ from its state at time t. Since any linear motion in three dimensions is reducible to motion along a single axis, this code learns to predict future states under optimal conditions. The samples' accelerations, velocities, positions are randomly generated, with the option to set acceleration to 0. NBSTE is the generalization of NSKTPC to two-dimensional states, but without acceleration.

NCEENMP tests a feedforward network's ability to predict two particles' post-collision velocities from their pre-collision velocities. The collision is elastic.

NCC accepts the initial states for two particles as input and predicts whether they will collide in the future or not.

Each code uses the Adam optimization algorithm[1] built into Keras with a learning rate of 0.001 (or 0.0005 in the case of NBSTE).

Table 1: Network Acronyms

Acronym	Full Name
NST	Net_Simple_Test.py
NSKTPC	Net_Single_Kinematic_Timestep_Plus_Conservation.py
NBSTE	Net_Bidirectional_Single_Timestep_Estimation.py
NCEENMP	Net_Collision_Endpoint_Estimation_No_Mass_Prediction.py
NCC	Net_Collision_Classification.py

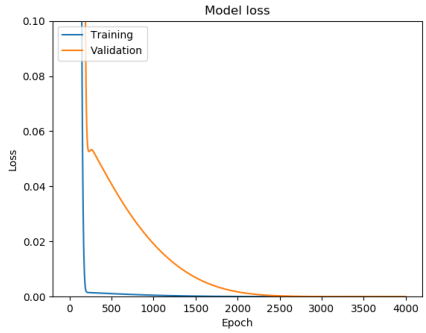
Table 2: Network Overview

Name	Purpose	Input	Output
NST	Learn identity, zero, alternating, and cubic	Two size 9 vectors	Two size 9 vectors
NSKTPC	1D next state x and v prediction; 1 particle	x_t, v_t, Δt, $a_{t+\Delta t}$	$x_{t+\Delta t}$,$v_{t+\Delta t}$, $a_{t+\Delta t}$
NBSTE	2D next state $\vec{x}$ and $\vec{v}$ prediction; 1 particle	$\vec{x}_t$, $\vec{v}_t$, Δt	$\vec{x}_{t+\Delta t}$, $\vec{v}_{t+\Delta t}$
NCEENMP	Predict $\{\vec{v}\}_{post}$ of elastic collision.	$\{\vec{v}\}_{pre}$ and $\{m\}$	$\{\vec{v}\}_{post}$
NCC	Predict whether state precedes collision	$\{\vec{x}\}_{pre}$,$\{\vec{v}\}_{pre}$, r	Binary

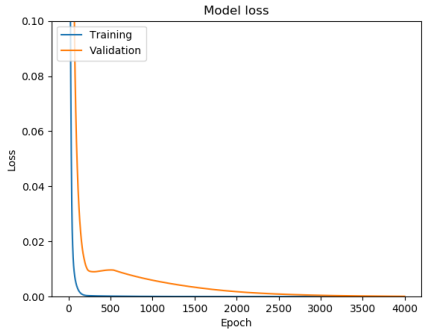
3 Results

These results and others are available on GitHub:
https://github.com/llfuller/NeuralNetworkNewton

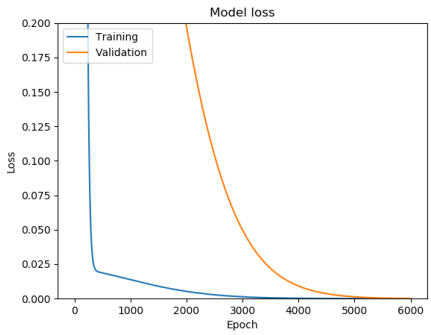
3.1 NST



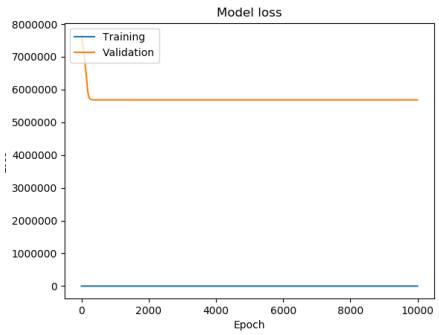
(a) Learning identity



(b) Learning zeros



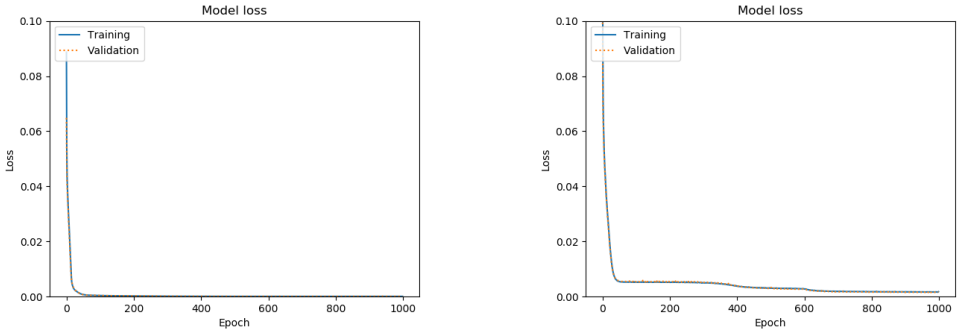
(c) Learning alternating pattern



(d) Learning cubic

Figure 2: NST mean squared error loss convergence for four functions with two LeakyReLU layers.

3.2 NSKTPC



(a) No acceleration (b) Acceleration

Figure 3: NSKTPC mean squared error loss convergence for 1,000 samples. The training and validation curves almost completely overlap.

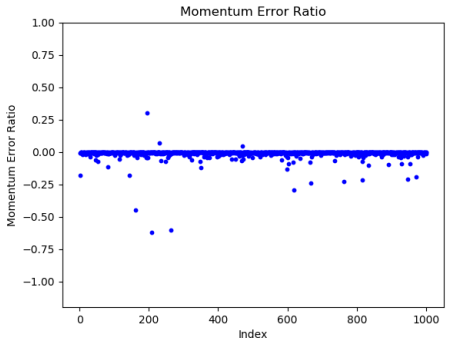


Figure 4: NSKTPC percentage momentum error for each trial. Error of 1 indicates 100% overestimation of true value.

3.3 NBSTE

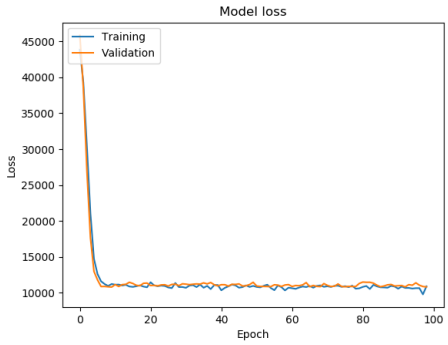
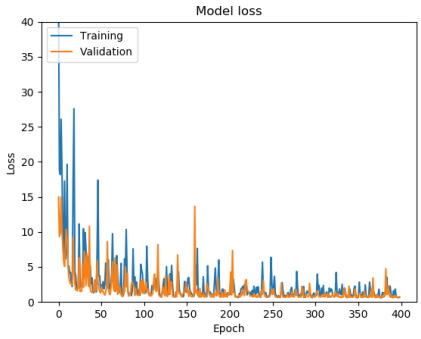


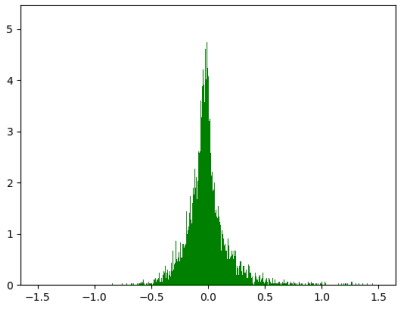
Figure 5: NBSTE loss convergence for 10,000 samples.

In the code PredictState.py, NBSTE failed spectacularly when tasked with predicting a time series of states recurrently from a single initial state.

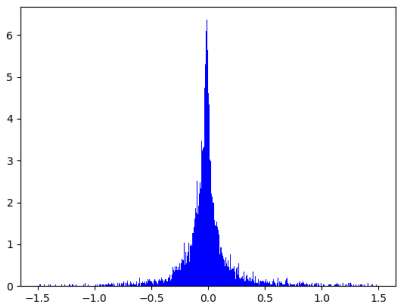
3.4 NCEENMP



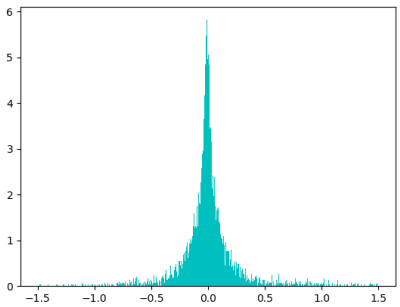
(a) NCEE loss convergence



(b) Energy error ratio



(c) X momentum error ratio



(d) Y momentum error ratio

Figure 6: NCEE loss convergence and conserved quantity error ratio in validation set of 10,000 samples. Error of 1 corresponds to 100% overestimation.

When calculating statistical quantities, I am excluding outliers, defined as samples with error over 200% in either direction, from consideration. This is justified by the shape of the distributions in Figure 6. At the end of the 400 epoch training period, the medians of the validation set's percentage errors are -2.3% for energy, -1.6% for x momentum , and -0.66% for x momentum. The amount of samples which are correctly predicted to have all three conserved quantities within 10% of the true value for that sample is 31.89% for this particular set of validation samples.

3.5 NCC

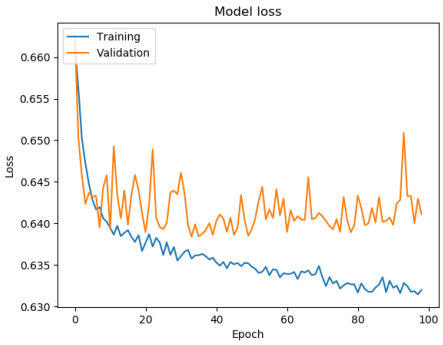


Figure 7: NCC loss convergence.

NCC on the validation set of 10,000 samples predicted 1,128 true collisions, 1,156 false collisions, 5,020 true misses, and 2,696 false misses.

4 Conclusions

The simpler patterns are straightforward for the feedforward neural networks to learn, but they do not learn easily in two spatial dimensions. NST learns all of its simple patterns well, except the network overtrains for cubic training samples and doesn't generalize well to the validation samples. This problem may be avoided by increasing the number of training samples. NSKTPC learns well with and without acceleration and shows that one dimensional kinematics is easy for feedforward networks to learn. NBSTE performs badly, possibly because the transition to two dimensions adds extra inputs and outputs and complicates the network. A network with rotational invariance which works as well as NSKTPC would solve this problem for NBSTE. NCEE learns to predict outgoing particle velocities in a manner consistent with elastic collision, but not very well. NCC demonstrates poor performance when predicting collisions from initial state data, and begins to overtrain after around 20 epochs. Considering all tests performed, I conclude that the densely connected feedforward networks are not suited to any aspect of the problem of collision with the current choice of loss, activations, depth and width. They are however able to predict one dimensional (accelerated and unaccelerated) motion well, and converge quickly for even simpler functions.

There were many results I encountered while developing these networks. In my experience with NCEE, length and width of the network in excess of a certain size didn't improve learning by a noticeable amount. At one point I had chosen a categorical cross-entropy loss function, which didn't allow the network to learn well. When the loss function was changed to mean absolute percentage error (and later mean squared error), the network was able to learn more easily. ReLU activation functions were problematic because they sometimes caused the outputs to freeze incorrectly at zero. This occurs because ReLU has zero gradient for non-positive values. Leaky ReLU activation functions eliminate this disadvantage. Another observation is that the number of training epochs for these networks may number in the hundreds or thousands before the loss function reasonably converges to a value near its long-term minimum (see Figure 2). Scale differences in variable magnitudes appear to make learning more difficult in feedforward networks, while numbers closer to one seemed to be easier to learn from. The PredictState.py code, which generates 2D motion recurrently from the NBSTE trained model, makes it clear that NBSTE is a poor predictor of motion in 2D. By testing in NSKTPC, I found the mean squared error loss slightly outperforms the mean absolute percentage error loss. This project taught me many valuable lessons about the limitations of primitive feedforward ANNs, and additional plots and codes from this project are available on GitHub for anyone interested in testing or building on it.

5 Future Direction

The most promising next step is to find or develop more advanced kinds of network architectures, such as LSTM, CNN, or attention networks which would learn from the data better than the presently used dense feedforward networks. The simulation output is sufficiently general to continue to be used with other learning algorithms, and it would be very easy to substitute another type of network into the analysis code. A CNN may work well for collision detection because all time points for each particle can be included in a single visual (as in Figure 1) and trained on similarly to MNIST data. LSTM could possibly help the network keep track of changes in direction which follow a collision. There is also the possibility of applying biological neural networks to these tasks. Future efforts most importantly must account for the diversity of physical interactions in the real world.

6 Acknowledgements

I would like to acknowledge Rafael Zamora-Resendiz for helping me debug the very first Keras network code in the project as well as suggesting and using attention networks for further study of the simulation data. I also give thanks to Professor Cauwenberghs and the neurodynamics TAs Margot and Jeremy for their useful advice and direction.

References

[1] Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980, 2017.*

7 Appendix

The neural network configuration and performance listed here for each test is the most effective (or nearly most effective) I found for that particular training assignment. All codes are available at https://github.com/llfuller/NeuralNetworkNewton

7.1 NST architecture

Remove LeakyReLU layers to get fully linear layers

```
Model: ``sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 9)                 90
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 9)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 9)                 90
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 9)                 0
=================================================================
Total params: 180
```

7.2 NSKTPC architecture

```
Model: ``sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 20
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 20
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 4)                 0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 20
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 4)                 0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 15
=================================================================
Total params: 75
```

7.3 NBSTE architecture

```
Model: ``sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 24
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 40
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 36
=================================================================
Total params: 100
```

7.4 NCEENMP architecture

```
Model: ``sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 6)                 42
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 6)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 48)                336
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 48)                0
_________________________________________________________________
dense_2 (Dense)              (None, 48)                2352
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 48)                0
_________________________________________________________________
dense_3 (Dense)              (None, 48)                2352
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 48)                0
_________________________________________________________________
dense_4 (Dense)              (None, 48)                2352
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 48)                0
_________________________________________________________________
dense_5 (Dense)              (None, 48)                2352
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 48)                0
_________________________________________________________________
dense_6 (Dense)              (None, 4)                 196
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 4)                 0
=================================================================
Total params: 9,982
```

7.5 NCC architecture

```
Model: ``sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 27)                270
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 27)                0
_________________________________________________________________
dense_1 (Dense)              (None, 27)                756
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 27)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 28
=================================================================
Total params: 1,054
```