Artificial Neural Networks and Applications - Precision Techniques

Ioannis Liodis[1], *Supervisor* Prof. Nikolaos Stergioulas [1]

Aristotle University of Thessaloniki[1]

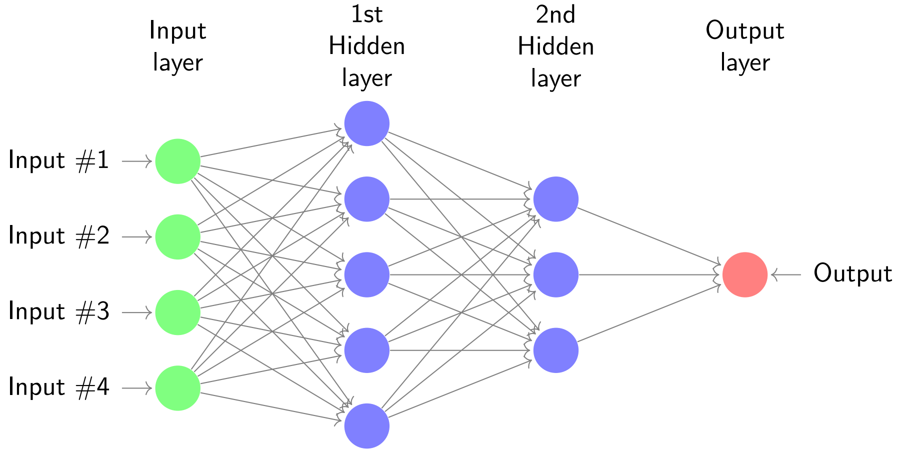
Bachelor Thesis
2022 - 2023

Outline

Outline

Structure

A neural network consists of multiple neurons, grouped in layers.



Training

The training requires a data set with the real inputs and outputs. This data set is divided into:

- **Training set** (usually 70% of the whole data set): These data are used to train the network, which means that in each epoch the goal is to minimise the loss between the real output values of the training set and the predicted values of the network.
- **Testing set** (usually 30% of the whole data set): These data are used to test the network after it is trained, which means that after all epochs the predicted values are compared to the real output values of the testing set.
- **Validation set** (usually 20% of the training set): These data are used to validate the network while it is being trained, which means that after each epoch the loss between the real output values of the validation and the predicted values of the network is calculated.

Note: The network is trained only by the training set.

Loss Function

Choose a loss function (for each network!):

$$L(y) = (y - y_{true})^2$$

Choose an activation function (for each layer!):

$$y = A\left(\sum_{i=0}^{3} w_i x_i\right)$$

We want to know the weight dependence on the loss function, in order to minimize the loss.

Loss Minimisation

So we calculate the partial derivatives:

$$\frac{\partial L}{\partial w_i}$$

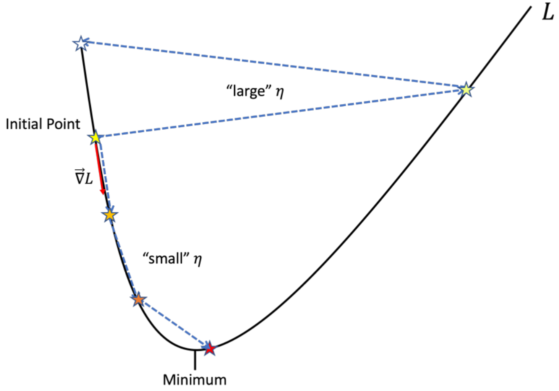
Back-propagation is the process of finding those derivatives. Then we proceed to updating the weights proportionally to their influence:

$$w_i \to w_i - \eta \frac{\partial L}{\partial w_i}$$

where η is called learning rate.

Loss Minimisation

The loss minimisation method is called gradient descent.



Note: We have to be careful when choosing η!

Gradient Descent - Important for later

Types of gradient descent (fixed learning rate):

- Stochastic gradient descent - STG
- Batch gradient descent
- Mini-batch gradient descent

Optimizer algorithms (flexible learning rate):

- Momentum
- Nesterov momentum
- Adagrad
- ADAptive momentum estimation - ADAM

The whole process

Structure steps:

- Architecture
- Choice of an activation function for each layer
- Choice of a loss function

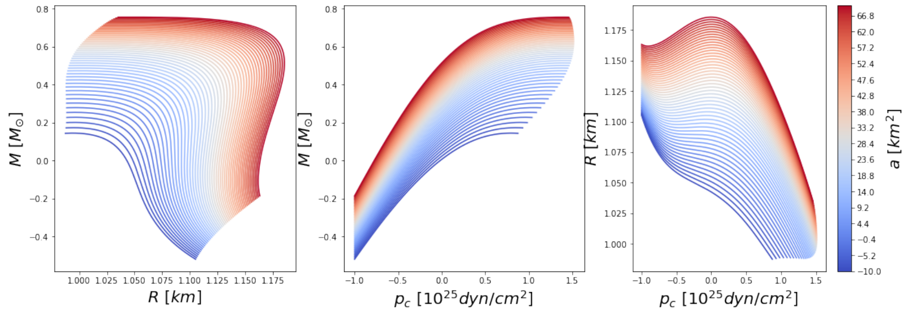
Training steps (one epoch):

- Random choice of $\vec{w}$
- Calculation of the output $A(\vec{x} \cdot \vec{w})$
- Back propagation and loss minimisation

Outline

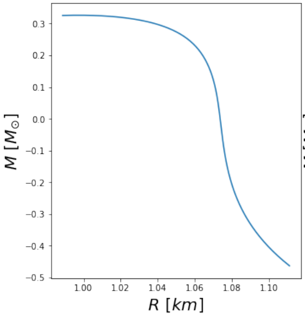
Data

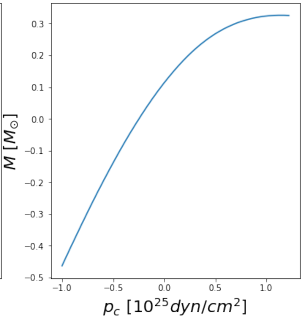
The whole data set used refers to the Einstein-Gauss-Bonnet Gravity for different values of α and p_c (51 $\times$ 200 and in log scale!) and for the SLy EOS.

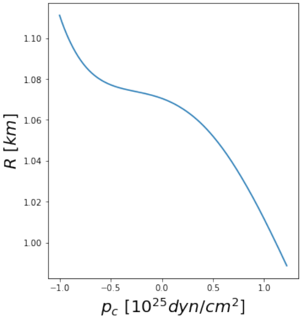


Data for examples

For each specific example I used an arbitrary choice of α to content to one curve. Here $\alpha = 1.2$:







Outline

Introduction

In scientific applications we attempt to fit Machine Learning models very precisely to data. Then some unique considerations arise and the this regime is called **Precision Machine Learning** (PML), according to Eric J. Michaud, Ziming Liu, Max Tegmark.

Note that the smallest nonzero difference allowed when fitting models to data is determined by the numerical precision used. Usually this is $2^{-52} \sim 10^{-16}$. Thus we should not expect a lower RMSE loss than 10^{-16}. Let $f_\theta(\vec{x}_i)$ be the model prediction to a data point y_i, on a dataset D. Then:

$$l_{rms} \equiv \left(\frac{\sum_{i=1}^{|D|} |f_\theta(\vec{x}_i) - y_i|^2}{\sum_{i=1}^{|D|} y_i^2} \right)^{\frac{1}{2}} = \frac{|f_\theta(\vec{x}_i) - y_i|_{rms}}{y_{rms}}$$

Loss decomposition

So on a dataset D we can define the emperical loss:

$$l_{rms}^{D} \equiv \left(\frac{\sum_{i=1}^{|D|} |f_\theta(\vec{x}_i) - y_i|^2}{\sum_{i=1}^{|D|} y_i^2} \right)^{\frac{1}{2}} = \frac{|f_\theta(\vec{x}_i) - y_i|_{rms}}{y_{rms}}$$

While on a probability distribution $\mathbb{P}_{(\mathbb{R}^d,\mathbb{R})}$ the expected loss:

$$l_{rms}^{\mathbb{P}} = \left(\frac{\mathbb{E}_{(\vec{x},y)\sim\mathbb{P}} \left[(f_\theta(\vec{x}) - y)^2 \right]}{\mathbb{E}_{(\vec{x},y)\sim\mathbb{P}} \left[y^2 \right]} \right)^{\frac{1}{2}}$$

Loss decomposition

The RMSE loss is not the only one we can define. For example:

- Relative MSE loss: $l_{mse} \equiv l_{rms}^2$
- Standard MSE loss: $L_{mse} = \frac{1}{|D|} \sum_{i=1}^{|D|} (f_\theta(\vec{x}_i) - y_i)^2$
- Standard RMS loss: $L_{rms} = \sqrt{L_{mse}}$

Loss decomposition

A given model architecture parametrizes a set of expressible functions $\mathcal{H}$:

- The best model on the expected loss $l^{\mathbb{P}}$

$$f_{\mathbb{P}}^{best} \equiv argmin_{f \in \mathcal{H}} \{ l^{\mathbb{P}}(f) \}$$

- The best model on the empirical loss l^{D}

$$f_{D}^{best} \equiv argmin_{f \in \mathcal{H}} \{ l^{D}(f) \}$$

- The model found by a given learning algorithm $\mathcal{A}$ which performs possibly imperfect optimization to minimize emperical loss L on D

$$f_{D}^{used} = \mathcal{A}(\mathcal{H}, D, L)$$

Loss decomposition

The different sources of error can be decomposed and termed as:

- **architecture error**: the best possible performance that a given architecture can achieve on the task
- **optimization error**: the error introduced by imperfect optimization
- *sampling luck*
- *generalization gap*

$$\ell^D(f_D^{\text{used}}) = \underbrace{[\ell^D(f_D^{\text{used}}) - \ell^D(f_D^{\text{best}})]}_{\text{optimization error}} + \underbrace{[\ell^D(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f_D^{\text{best}})]}_{\text{sampling luck}} + \underbrace{[\ell^{\mathbb{P}}(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f_{\mathbb{P}}^{\text{best}})]}_{\text{generalization gap}} + \underbrace{\ell^{\mathbb{P}}(f_{\mathbb{P}}^{\text{best}})}_{\text{architecture error}}$$

Remark: The dominant error source determines the empirical loss!

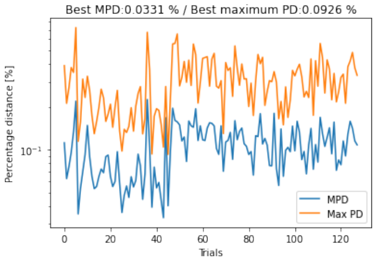
Outline

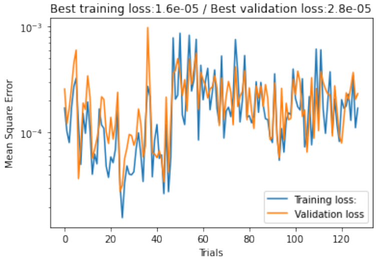
Best architecture

The best activation function combination (using Adam optimizer and for one M-R curve) were:

- 3 hidden layers: tanh - relu - tanh
- 5 hidden layers: tanh - relu - tanh - relu - tanh

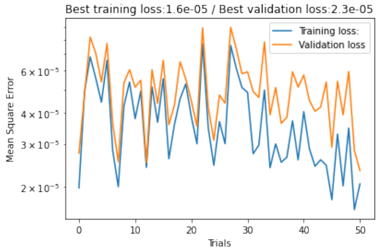
For these models, a more systematic investigation was done concluding on the best model to be the one with 5 hidden layers and 25 - 35 - 45 - 35 - 25 neurons.

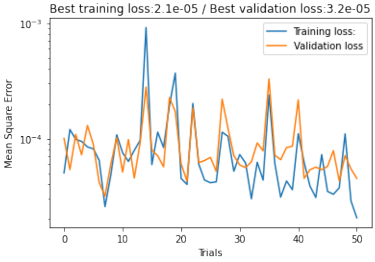




Best architecture

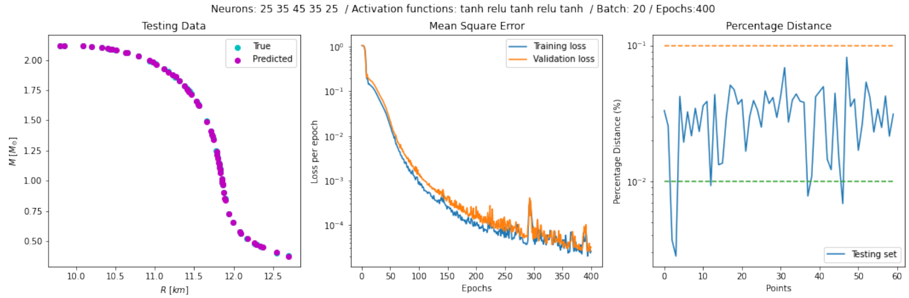
An interesting result is that the same model is fitting **every M-R curve** with almost the same accuracy (but always quite unstable).





Best architecture

An indicative fit of the best architecture.



The results are not optimal. Obviously, the architecture error is not the dominant source.

Outline

Optimizers

Optimizers are algorithms with flexible learning rates that can avoid the problems imposed by a fixed learning rate, as discussed in the previous presentation.
Some algorithms are:

- Momentum
- Nesterov momentum
- Adagrad
- ADAptive momentum estimation - ADAM
- Adamax
- Nadam

Momentum

Simile between the position on the parameter space ($\vec{w}$) and the position on a geographical analog ($\vec{r}$). The position update is proportional to the momentum! The update at each step also depends on the previous update, making it a recursive sequence:

$$\vec{u}_t = \gamma \vec{u}_{t-1} + \eta \nabla L_{(\vec{w})}$$

$$\vec{w}' = \vec{w} + \vec{u}_t$$

where t is the current update and γ is the parameter controlling the momentum (usually $\gamma = 0.9$).
Note: The learning rate is fixed.

Nesterov Momentum

The difference is that it uses the momentum to approximate the next point and calculate the gradient there:

$$\vec{u}_t = \gamma\vec{u}_{t-1} + \eta\nabla L_{(\vec{w}+\gamma\vec{u}_{t-1})}$$

$$\vec{w}' = \vec{w} + \vec{u}_t$$

$$\vec{w}' = \vec{w} + \vec{u}_t$$

Note: The learning rate is fixed.

Adaptive gradient - Adagrad

This algorithm updates each parameter based on all the past values and uses a different learning rate for each parameter at each step:

$$\eta_{i,t} = \frac{\eta}{\sqrt{G_{i,t} + \epsilon}}$$

$$G_{i,t} = \sum_{k=0}^{t-1} (\nabla L_{(w_{i,k})})^2$$

where i refers to the different weights and ϵ is a very small number to avoid division with zero. Then:

$$w_{i,t+1} = w_{i,t} + \eta_{i,t} \nabla L_{(w_{i,t})}$$

ADAptive Momentum estimation - ADAM

This algorithm uses decaying average of gradients and at the same time implements a similar technique to momentum. In this case the gradients are treated as random values and the moments are defined as the expectation values of them to a power. Defining g as the gradient, then the first two moments are the mean and the variance respectively:

$$m_{i,t} = \beta_1 m_{i,t-1} + (1-\beta_1) g_{i,t} \quad \& \quad u_{i,t} = \beta_2 u_{i,t-1} + (1-\beta_2) g_{i,t}^2$$

where β_1, β_2 are commonly fixed to 0.9 and 0.999 respectively. The initial values $m_{i,0}, u_{i,0}$ are set to be zero.
In order to remove the bias (it can be proven):

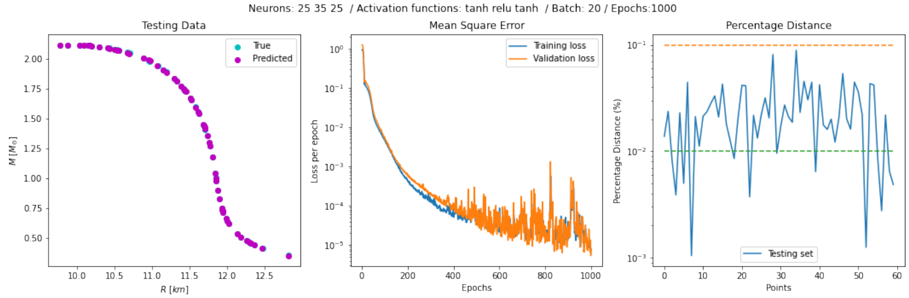
$$\hat{m}_{i,t} = \frac{m_{i,t}}{(1-\beta_1^t)} \quad \& \quad \hat{u}_{i,t} = \frac{u_{i,t}}{(1-\beta_2^t)}$$

Then:

$$w_{i,t+1} = w_{i,t} - \hat{m}_{i,t+1} \frac{\eta}{\sqrt{\hat{u}_{i,t+1} + \epsilon}}$$

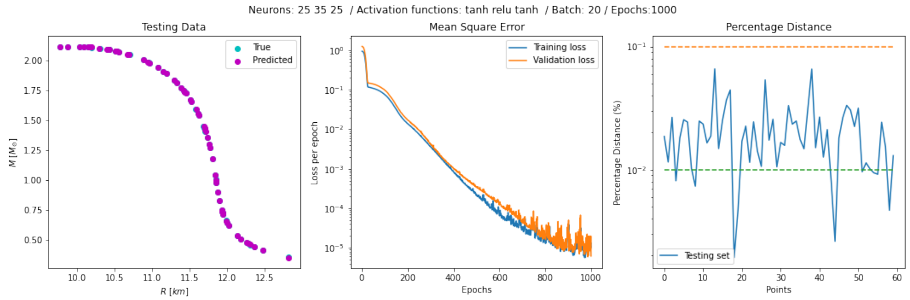
Examples with built tensorflow optimizers

Adam
MPD : 0.024117 %
MAPE : 0.073819 %



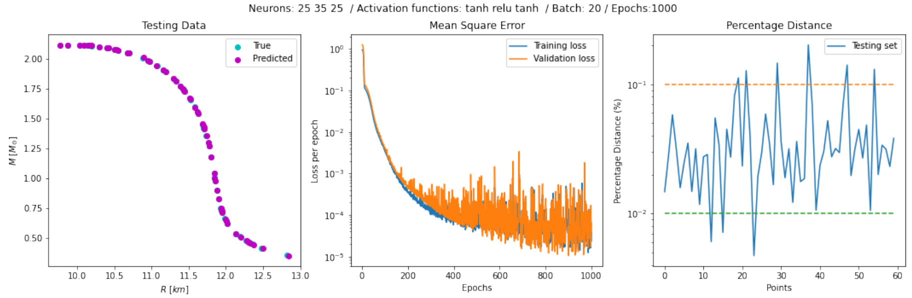
Examples with built tensorflow optimizers

Adamax
MPD : 0.020531 %
MAPE : 0.046021 %



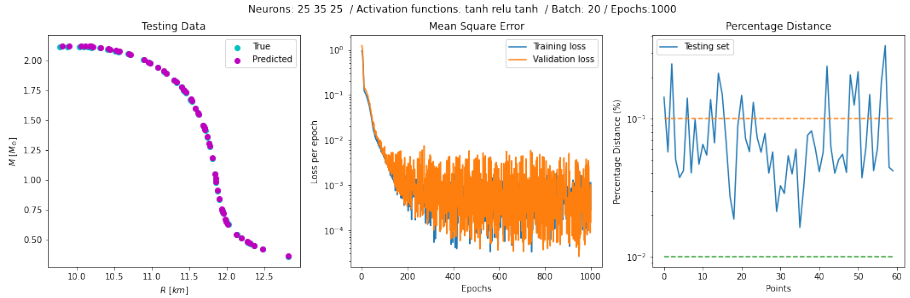
Examples with built tensorflow optimizers

Nadam
MPD : 0.041871 %
MAPE : 0.122074 %



Examples with built tensorflow optimizers

RMSdrop
MPD : 0.085209 %
MAPE : 0.236790 %



Note: every optimizer until now calculates only the first derivative! These algorithms are called **first order optimizers**.

Newton's method

From the numerical analysis course we know that in order to find a local minimum for a function f of one variable we can use the Newton's method:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This can be generalised using the Hessian matrix:

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Newton's method

And the generalised iteration step is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [H(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k)$$

Although this method is very accurate (since it exploits information of 2nd order derivatives), it is also computationally expensive (computing the inverse Hessian scales as $O(n^3)$).
There is a class of optimization methods that attempt to address this issue, which is called **Quasi-Newton methods**.
Broyden - Fletcher - Goldfarb - Shanno (**BFGS**) optimization is one of the most popular such methods (2nd order optimizer).

For details read this article.

BFGS - Problem

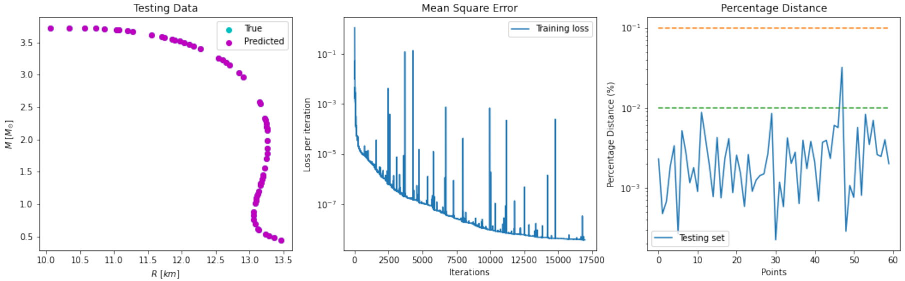
The only problem with this optimizer is that it is not built in like the previous optimizers which could be chosen when compiling.

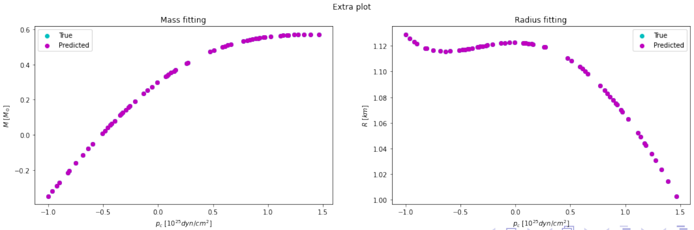
It is built in tensorflow probability and can be easily implemented in training neural networks using this code either for BFGS or L-BFGS.

BFGS - Implementation to one curve

MAPE M: 0.003863 % MAPE R: 0.000793 %

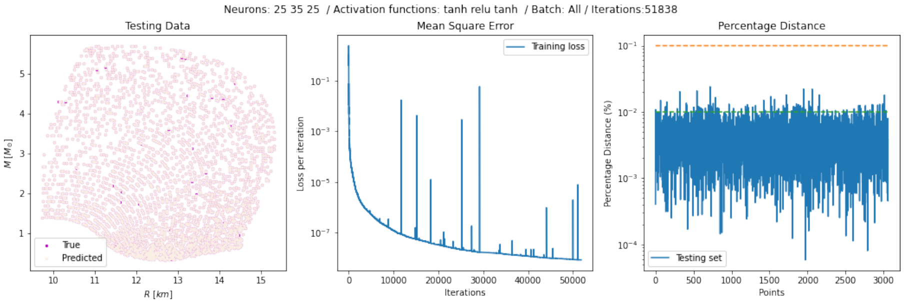
Neurons: 25 35 25 / Activation functions: tanh relu tanh / Batch: All / Iterations:16976

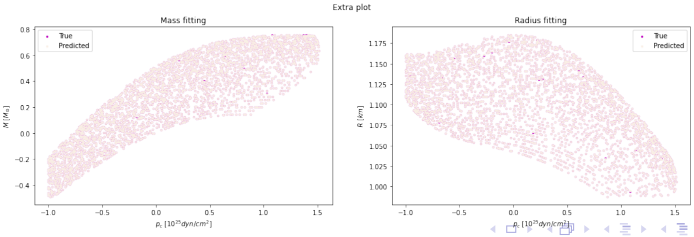




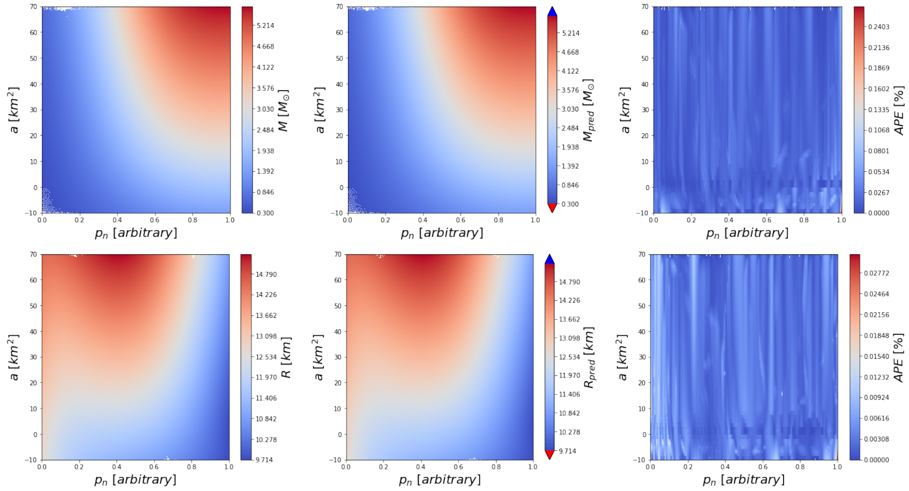
BFGS - Implementation to all curves

MAPE M: 0.004553 % MAPE R: 0.000943 %





BFGS - Implementation to all curves



Note: the data are given randomly!

Outline

Low-Curvature Subspace Optimization

Restricting optimization to low-curvature subspaces the loss can be further decreased. This method has a single hyperparameter τ:
Let $g = \nabla_\theta \mathcal{L}$ be the gradient and H be the Hessian of the loss. Instead of heading towards the direction $-g$, instead head towards $-\hat{g}$, where

$$\hat{g} = \sum_{i:\lambda_i<\tau} e_i(e_i \cdot g)$$

This method improves the RMSE loss by a factor of over $2x$.

Note: I did not try this yet.

Boosting

Boosting is a method that uses two neural networks trained sequentially: let f_{θ_1}, f_{θ_2}. The first is trained to fit the target f, and the second is trained to fit the residual $\frac{f-f_{\theta_1}}{c}$, where $c << 1$ normalizes the residual to be of order unity. Finally their outputs are added as:

$$g = f_{\theta_1} + cf_{\theta_2}$$

This method can lower the RMSE loss further by 5-6 orders of magnitude.

Note: I tried this, but it did not work.

Boosting - Problem

There are several possible issues:

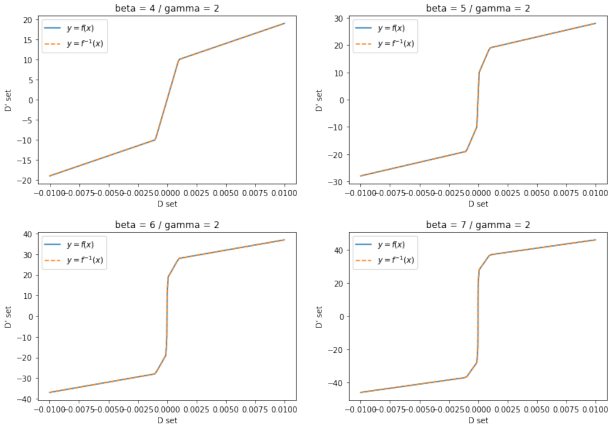
- Not all the residuals are of the same order (i.e. $|dR| \in [10^{-8}, 10^{-3}]$), so what will c be?
- Standardization is useless.
- Cannot use log because I need to keep the sign.

Ideas for solving these issues:

- Try an other "1-1" non linear map $D \to D'$.
- Standardize the new D'.
- Try to mimic log but use an odd function.

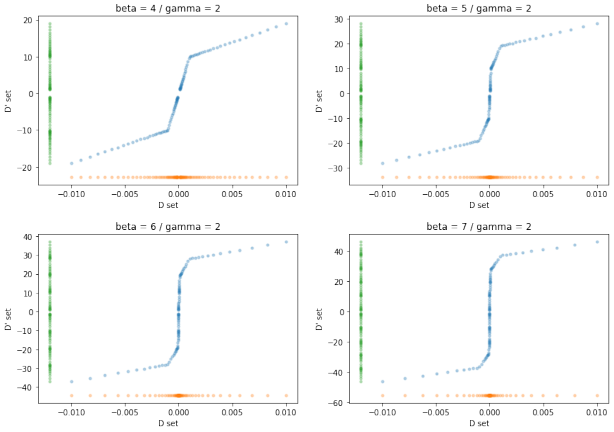
Boosting - Suggestion

A function with the above properties is a piecewise linear, monotonous function which maps each order of magnitude of D to a unit/decade of D'.



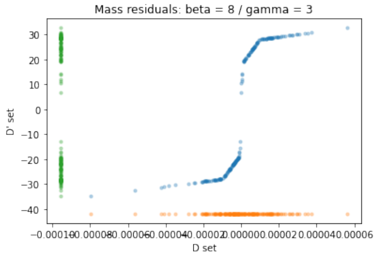
Boosting - Suggestion

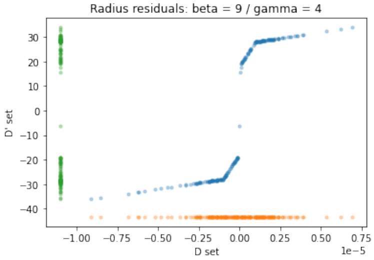
Hence, D' is more sparse, ready to be Standardized and to train (here D consists of two symmetric logspaces).



Boosting - Suggestion

The implementation on real residuals:



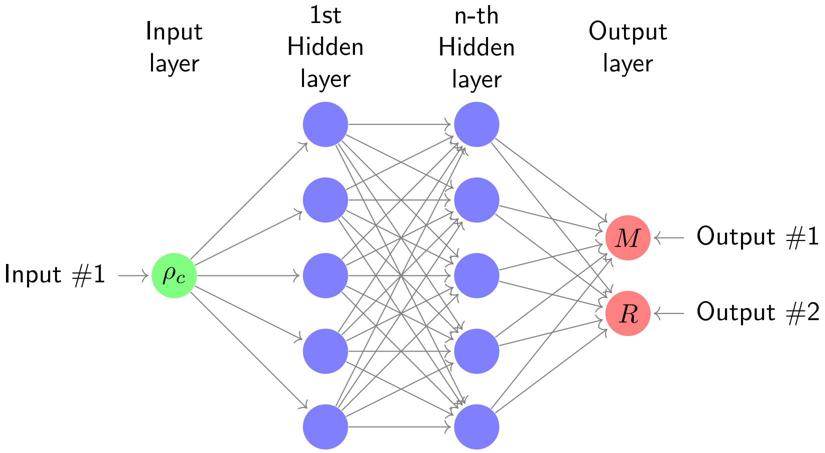


Question: Should I try with different slopes? Note: There is an issue with the inverse.

Outline

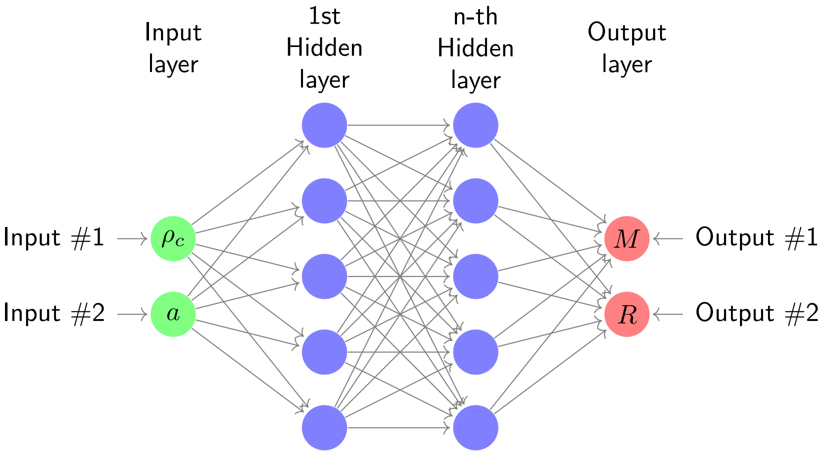
General Relativity (Done with great precision)

Up to this point we had one input and two outputs, and one network for every EOS. This corresponds to a **curve**.



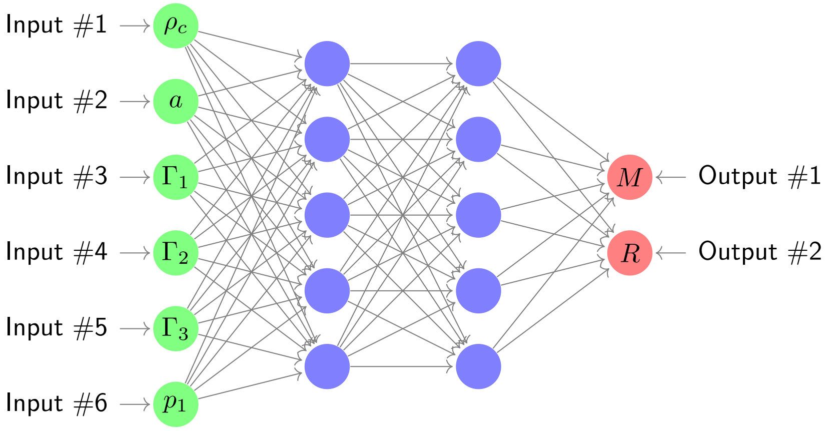
Einstein-Gauss-Bonnet Gravity (Done with good precision)

The first target is to add one more input parameter, which parameterizes how "far" the modified gravity is from GR (one network for every EOS). This corresponds to a **surface**.



Einstein-Gauss-Bonnet Gravity (Not done yet)

The second target is to add 4 more input parameters, except a, which parameterize the EOS. This corresponds to a **surface**?



Outline

Near future work

For the 1-2 case:

- Try the BFGS from scipy.
- Implement the Low-Curvature Subspace Optimization.
- Implement boosting.

For the 2-2 case:

- Curriculum learning (train on each curve).
- Implement the 1-2 techniques.
- Try the same model in other EOSs.

Question: What limits should I impose in the 2-2 case?

Thank you!

Sources

- Precision Machine Learning, Eric J. Michaud, Ziming Liu, Max Tegmark https://arxiv.org/abs/2210.13447
- Prof. Nikolaos Stergioulas' presentation at 11th Aegean Summer School, Syros - Machine Learning Applications in Gravitational Wave Astronomy
- Vasileios Skliris PhD - Machine Learning To Extract Gravitational Wave Transients - Cardiff
- https://towardsdatascience.com/
- https://thinkingneuron.com/
- https://texample.net/
- https://www.tensorflow.org/
- https://github.com/niksterg/pyTOVtab
- https://gist.github.com/piyueh/712ec7d4540489aad2dcfb80f9a54993