Regression with Dense Neural Network using TensorFlow

Simone Albanesi, Marina Berbel

January 31, 2022

1 Introduction

Working notes on the regression part with a dense Neural Network (NN) using `Keras` with `TensorFlow` in back-end. We use this framework since allows us to exploit the flexibility of NNs (while `Scikit-learn` doesn't).

The task is to apply the NN to the quantities recovered from `GstLAL` x and to predict values $y^{\text{pred}} = f(x)$ that are a better approximation of the injected values y^{true}.

Some pros of NNs are:

- virtually infinite number of architectures, many aspects to tune on our problem
- they are used everywhere nowadays, so it is easy to find material online
- is possible to put constraints on the output
- easy to customize

On the other hand, to fully exploit the NN we need a big dataset and the high number of tunable aspects makes the cross validation more difficult. In our case we will use at most 2 hidden layers. In principle more complex architectures could be used but in our case I don't think it would be worth since we do not have a huge training sample. For regression tasks, the activation function in the output layer generally is a linear function.

Steps of the regression:

1. **Normalization of the data**. The usual normalization (`StandardScaler`) trasforms the data so that we have zero mean and standard deviation equal to 1. Nonetheless I prefer to use the linear map `MinMaxScaler` that maps the data in the interval $[-1, 1]$. This choice is linked to the constraints on the output, see Sec. 2.
2. **Initialization of the model**. We decide the architecture and all the other aspects. A typical configuration is:
   - layers: two layers with 100 neurons, but e.g. on the `v0c0` dataset a more complex architecture could be slightly better (see Sec. 4 for more info on this);
   - activation function in hidden layers: `ReLU`. Other options like the sigmoid or the hyperbolic tangent require longer training and provide worse results.;
   - activation function in output layer: `linear`;
   - optimizer: `Adam` (optimized version of the stochastic gradient descent);
   - loss function: `MeanSquaredError()` (see Sec. 6 for more details and other options);
   - learning rate: 0.001;
   - epochs and batch-size: a good combination is 250 and 32, respectively. Sometimes I use a bigger batch-size to speed-up the training (e.g. I used 128 for the cross-validation on the `v0c0` dataset).
3. **Training**. Using the dataset `v0c0`, $N_{\text{epochs}} = 250$ and $N_{\text{batch}} = 128$ the training takes from 60 to 90 seconds, depending on the architecture. During the training we also use the 20% of the training dataset for the validation so that we can monitor the loss and the R^2 coefficient (see definition below) at each epoch.
4. **Evaluation of the accuracy**. To evaluate the goodness of the model we use the coefficient of determination $R^2 = 1 - SS_{\text{res}}/SS_{\text{tot}}$, where $SS_{\text{res}} = \sum (y_i^{\text{true}} - f(x_i))^2$, $SS_{\text{tot}} = \sum (y_i^{\text{true}} - \bar{y}^{\text{true}})^2$, and $\bar{y}^{\text{true}}$ is the mean. This is also the `score` of `Scikit-learn`. It is better

to evaluate the R^2 coefficient for each feature instead of a global R^2.

2 Constraints on the output

We need some constraints on the output since we don't want to predict negative masses or naked singularities. A solution is to modify the activation function σ in the output layer. Another option could be to enforce the constraints in the loss function, but I have not tried this (and in any case this would not guarantee physical predictions). The idea is to make σ saturate when the prediction is out of the physical range. The first step is to normalize the data in the interval $[-1, 1]$ using the `MinMaxScaler`. Then for σ we use the following prescription:

$$\hat{\sigma}(x) = \begin{cases} -1 & x \leq -1 \\ x & -1 \leq x \leq 1 \\ 1 & x \geq 1 \end{cases} \tag{1}$$

Using a sigmoid ansatz is possible to find a smooth approximants

$$\sigma^N(x) = \frac{2}{e^{-2f^N(x)} + 1} - 1, \tag{2}$$

where

$$f^N(x) = \sum_{n=odd}^{N} \frac{x^n}{n}. \tag{3}$$

The function $f^N(x)$ is simply found requiring that $\sigma^N(x) = x + O(x^{N+2})$ for $x \to 0$. These activation functions are shown in the left panel of Fig. 1. However, for $N \geq 3$ is possible that some NaN appear during the training, see e.g. the right panel of Fig. 1 where we show a case with $\sigma^5(x)$. I remember that once also $\sigma^3(x)$ failed, but I cannot find this configuration again (is a rare event). In any case to be safe I will use the piecewise function $\hat{\sigma}$ ($\sigma^1(x)$ is not steep enough). Clearly it is also possible to saturate only the upper of the lower boundary, in particular we for the masses I will often use

$$\hat{\sigma}_{\rm LB}(x) = \begin{cases} -1 & x \leq -1 \\ x & -1 \leq x. \end{cases} \tag{4}$$

However I am not sure that removing the upper boundary makes sense, maybe it would be better to use the upper constraint also for the masses, TBD.

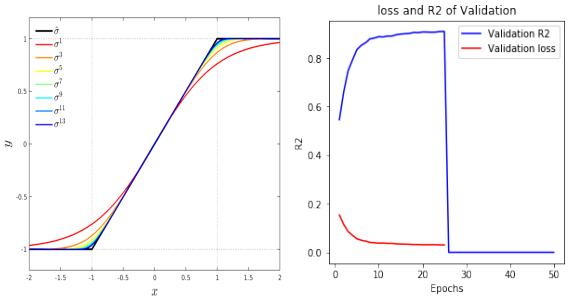

Figure 1: **Left panel**: activation functions to use in the output layer so that the output is constrained in the interval $[-1, 1]$. **Right panel**: history of a NN with two hidden layers with 20 neurons and $\sigma^5(x)$ as output function. The drop in the accuracy at the 18th epoch is due to the presence of NaN. That's why we use $\hat{\sigma}$ and not a smooth approximant σ^N.

3 Some results

The regression with NN works pretty well on the `v0` datasets, while on the dataset `v1` the R^2 coefficients are smaller since there is more degeneracy. We use 2 hidden layers with 100 neurons (i.e. 12,411 trainable parameters, while $N_{\rm sample} = 2 \cdot 10^4$) and we train on 250 epochs using $N_{\rm batch} = 64$. Different architectures are discussed in Sec. 4 but the results are more or less the same for complex-enough NNs. For the spin components we use $\hat{\sigma}$ as output activation function, for the masses and the mass ratio we use $\hat{\sigma}_{\rm LB}$ and a linear function for the angle. In Fig. 2, Fig. 3 and Table 1 there are the results for this model. In the first plots we plot the predicted value against the injected values, while in the seconds we plot the prediction (orange) on the recovered (blue). Note that the recovered here are not physically consistent but it is ok since the main goal at this point is to see if a neural network is able to recover the original quantities. The relevant thing is that we are able to predict values that have physical meaning.

4 Cross validation on layers

In order to decide the architecture to use I did a cross validation on the `v0c0` dataset (the reason why I have not cross-validated on the `GstLAL`

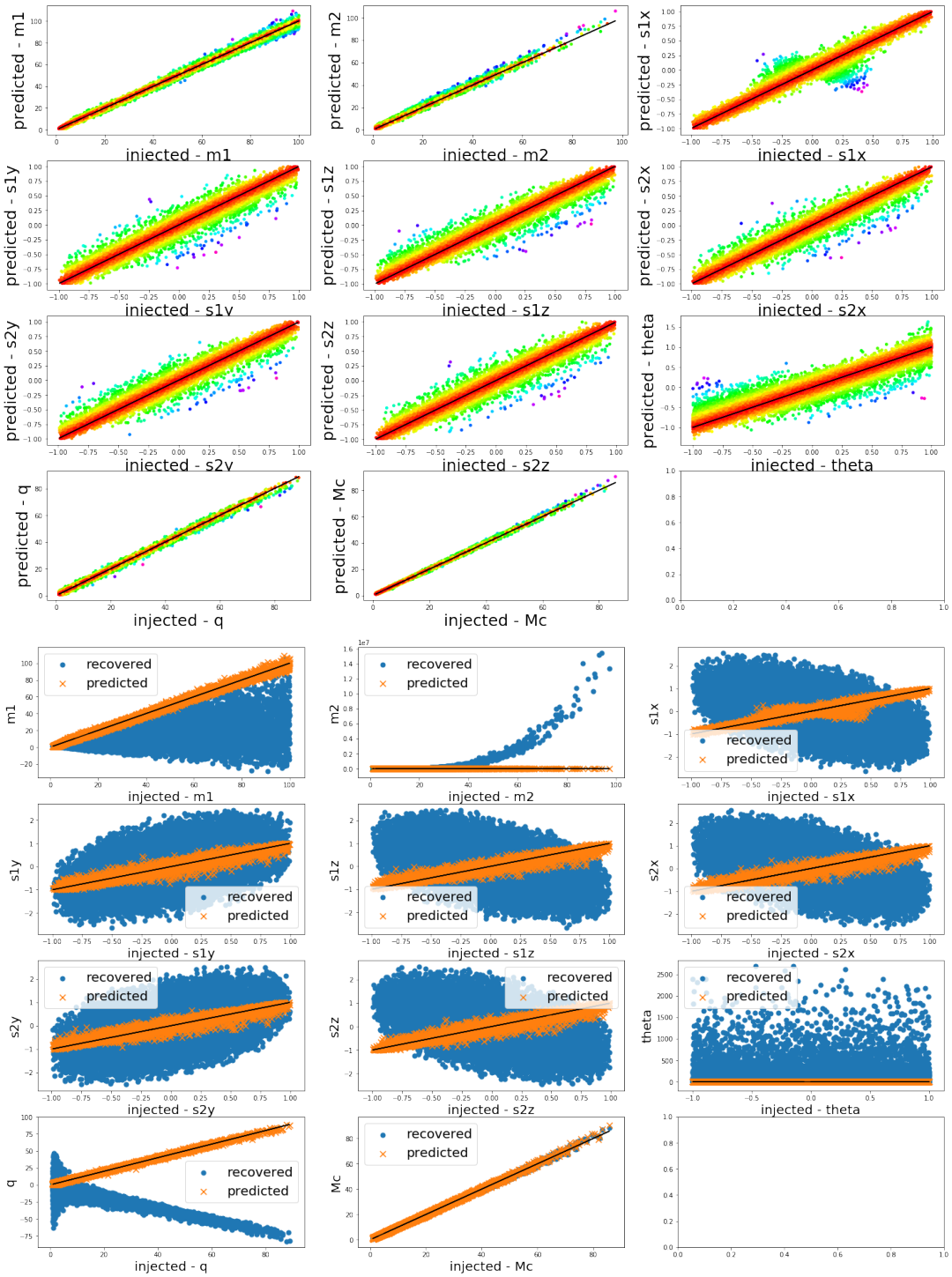

Figure 2: Results of the regression on the `v0c0` dataset. We used 2 hidden layers with 100 neurons, $N_{\text{batch}} = 64$, 250 epochs, constrained output. See Sec. 3 for discussion. The colors in the first plots is related to the absolute difference between predicted and injected. The black line is the bisector.

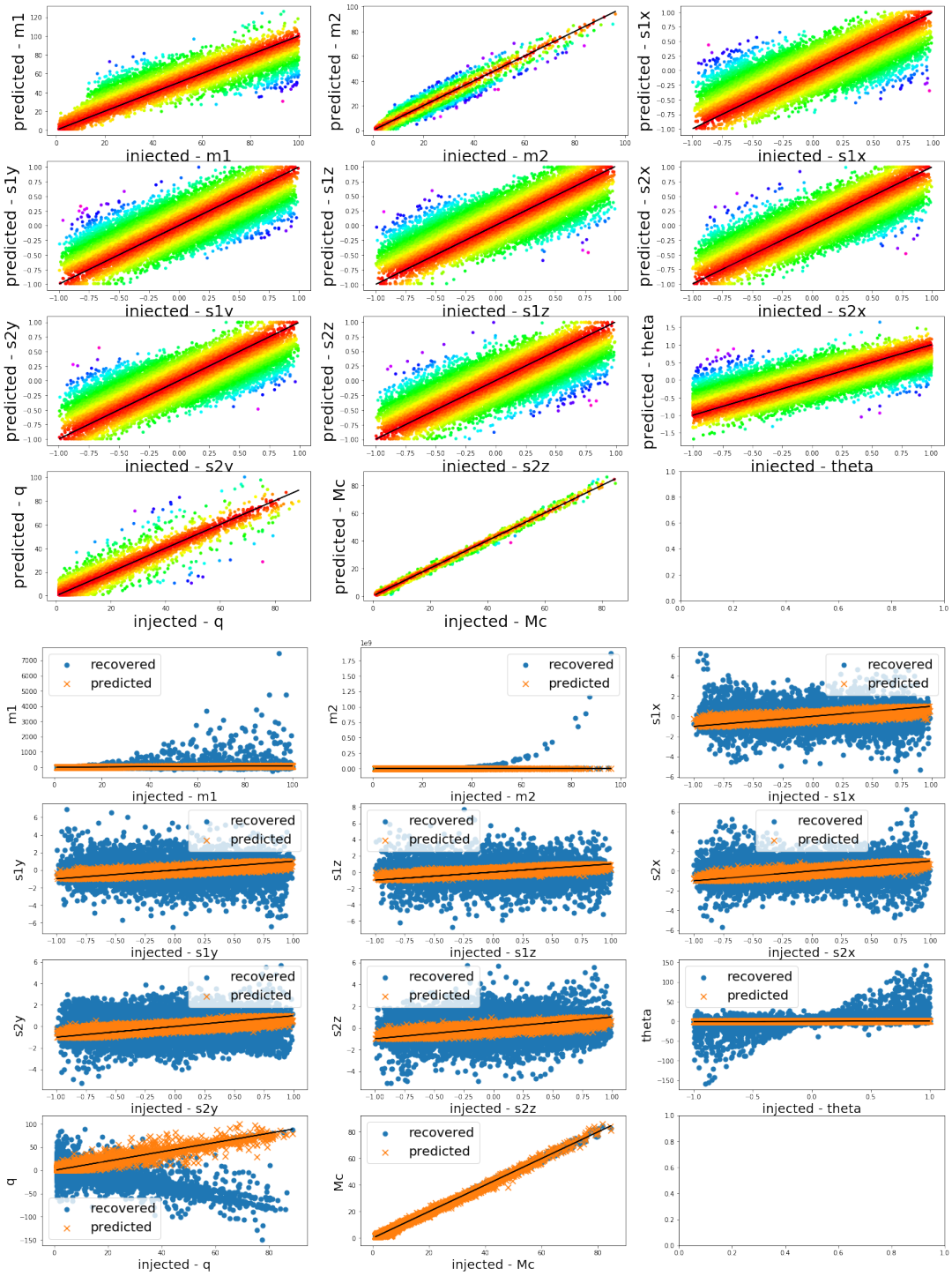

Figure 3: Results of the regression on the `v1c0` dataset. We used 2 hidden layers with 100 neurons, $N_{\text{batch}} = 64$, 250 epochs, constrained output. See Sec. 3 for discussion. The colors in the first plots is related to the absolute difference between predicted and injected. The black line is the bisector.

Table 1: Loss (Mean Squared Error) and R^2 coefficients for the two datasets `v0c0`, `v1c0`. We used 2 hidden layers with 100 neurons, $N_{\rm batch} = 64$, 250 epochs, constrained output. See Sec. 3 for discussion.

	`v0c0`	`v1c0`
loss (MSE)	0.0079	0.0533
R^2 mean	0.9652	0.7533
$R^2[m_1]$	0.9962	0.8938
$R^2[m_2]$	0.9860	0.9454
$R^2[s^1_x]$	0.9582	0.6134
$R^2[s^1_y]$	0.9529	0.6344
$R^2[s^1_z]$	0.9549	0.6436
$R^2[s^2_x]$	0.9539	0.6853
$R^2[s^2_y]$	0.9570	0.6409
$R^2[s^2_z]$	0.9568	0.6468
$R^2[\theta]$	0.9073	0.6648
$R^2[q]$	0.9957	0.9244
$R^2[M_c]$	0.9978	0.9938

dataset is explained in Sec. 5). I trained the models on 250 epochs using $N_{\rm batch} = 128$ for different architectures. The results are shown in Fig. 4. The plots show that NNs with 2 layers with ~ 100 neurons each provide good enough scores and more complex NNs do not provide much better results. After a certain number of trainable parameters the R^2 coefficient reaches a plateaux and doesn't increase any more. The only exceptions are: (i) M_c: the best score is obtained with single-layer NNs (the recovered quantity is obtained only adding random Gaussian noise); (ii) θ: this is the quantity with lowest score and the R^2 coefficient continues to (slightly) increase without reaching a plateaux even for ~ 300 neurons in each layer.

Finally note that many NNs have more parameters than training samples, $N_{\rm param} > N_{\rm train}$, since $N_{\rm train} = 2{\cdot}10^4$ for the `v0v0` dataset. This shouldn't be a problem for NN (and indeed the score is good also on the test-set that has $N_{\rm test} = 1.5 \cdot 10^4$), however we could also decide to use smaller NNs since the accuracy in more complex NNs is not drastically better than in simpler models. For example in Fig. 2 and Fig. 3 I used two hidden layers with 100 neurons so that $N_{\rm param} = 12411 < N_{\rm sample} = 2 \cdot 10^4$.

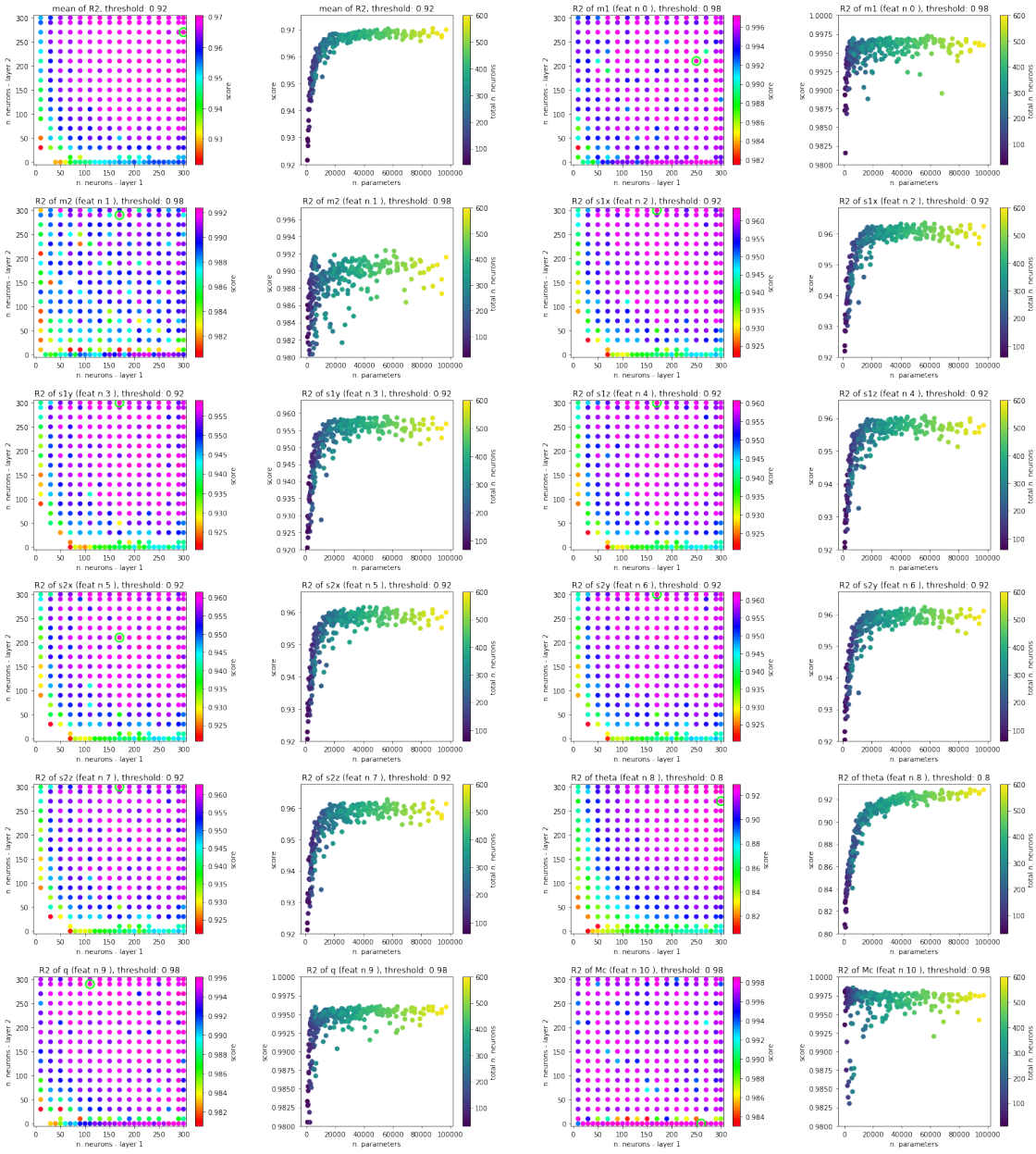

Figure 4: R^2 coefficients for different architectures. We show only the models that have an R^2 coefficient above the threshold indicated in the title of each panel. We show the results for the mean of R^2 and for the R^2 of each feature. In the rainbow-scatter plot the green circle marks the highest R^2 coefficient.

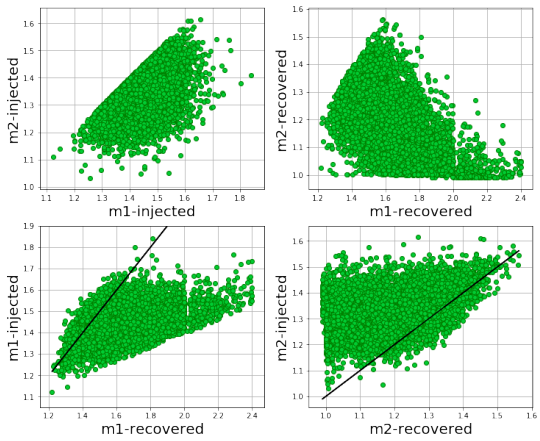

Figure 5: Test-dataset, compare with Fig.1 of Chatterjee-1911.00116.

5 GstLAL BNS dataset

We also have a dataset obtained with injected parameters of nonspinning BNSs recovered with `GstLAL`. In this case we have only three features: m_1, m_2, M_c. The regression on this dataset is more problematic:

- The output of the regression for the two masses m_i seems almost independent on the architecture and is not improved with more complex NNs. The result for M_c instead is more sensible on the architecture (but the regression seems useless for M_c)
- The predictions of m_i have a sharp edge in correspondence of the region with most degeneracy, see Fig. 6
- The model stops to learn after the second epoch. Training for more epochs only improves the prediction of the lowest and highest chirp masses, but does not improve the prediction of m_i.
- The mass ratio computed with the predicted masses is in a very narrow range while the injection have a much wider interval, see the last panel of Fig. 6.

Therefore in this case the regression does not seem to work properly. I have also tried to do the regression only on $m_{1,2}$ and M_c and recovering the other mass analytically but the results are the same. Even SVR gives the same results (qualitatively).

6 ~~Loss function~~ [wrong way of doing things, ignore me]

In order to fix the problem with q in the `GstLAL` dataset, I tried to modify the loss function to minimize. The default function for regression tasks is the Mean Squared Error

$$J_{\rm mse} = \text{mean} \sum_i \left(y_i^{\rm true} - f(x_i)\right)^2 . \tag{5}$$

I tried to modify it including a q-penalty and a $\mathcal{M}_c$-penalty:

$$\begin{aligned} J = \text{mean} \Bigg(& \sum_i \left(y_i^{\rm true} - f(x_i)\right)^2 \\ & + \lambda_{\rm q} \sum_i \left(q_i^{\rm true} - q_i^{\rm pred}\right)^2 \\ & + \lambda_{\mathcal{M}_{\rm c}} \sum_i \left(M_{c,i}^{\rm true} - M_{c,i}^{\rm pred}\right)^2 \Bigg) \end{aligned} \tag{6}$$

where (not surprisingly)

$$q_i^{\rm pred} = \frac{m_{2,i}^{\rm pred}}{m_{1,i}^{\rm pred}}, \tag{7}$$

$$M_{c,i}^{\rm pred} = \frac{\left(m_{1,i}^{\rm pred} m_{2,i}^{\rm pred}\right)^{3/5}}{\left(m_{1,i}^{\rm pred} + m_{2,i}^{\rm pred}\right)^{1/5}}, \tag{8}$$

and $q^{\rm true}$, $\mathcal{M}_c^{\rm true}$ are analogous. Note the $\mathcal{M}_c$-term is not the same in MSE since here the chirp mass is computed using m_i while in the MSE part $\mathcal{M}_c$ is the third feature. In order to have the predicted masses in the loss function we need to rescale x but since the loss function is defined using the Keras backend and for the moment this is implemented only for `MinMaxScaler`. We cannot use `numpy` in the Keras backend, so the loss function is not fully vectorized (but this is not a big issue). I tried to fit the data with different values for the hyperparameters λ_q and $\lambda_M c$ but in every case I didn't find a solution to the issues of Sec. 5 since the effect of these penalty is marginal, even if we use high values for λ_i. Maybe this is not the correct way to include penalties. UPDATE: indeed, this is wrong because the penalty is not included in the derivative.

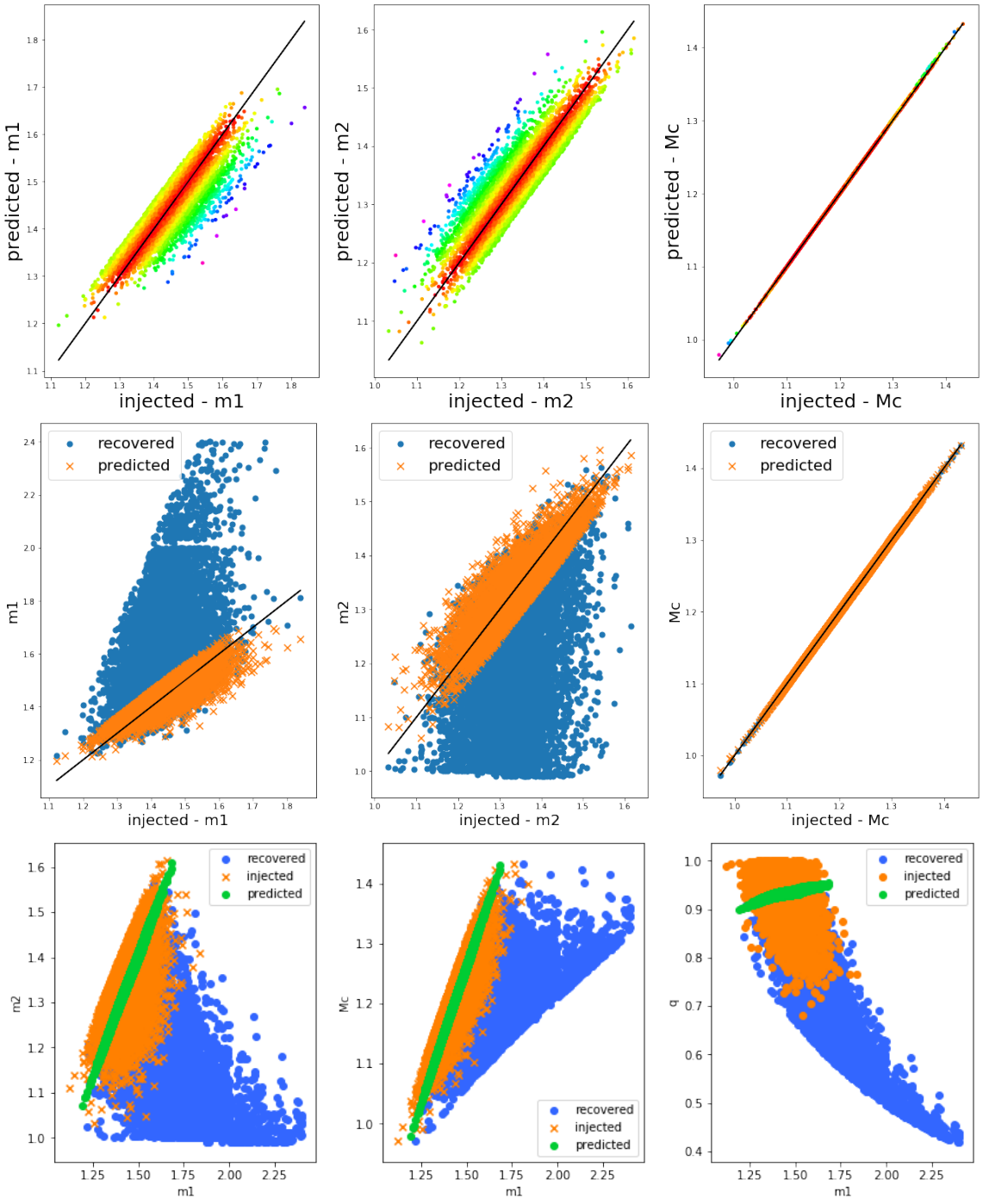

Figure 6: Results of the regression on the `GstLAL` dataset. We used one hidden layer with 100 neurons, $N_{\text{batch}} = 64$, 50 epochs, unconstrained output. The colors in the first plots is related to the absolute difference between predicted and injected. The black line is the bisector. The final R^2 coefficients for m_1, m_2 and M_c are 0.7363, 0.7819, and 0.9999, respectively. In the last three panels we plot m_2, $\mathcal{M}_c$ and q over m_1 for the injected, recovered and predicted quantities.

7 GstLAL BNS dataset - a different approach

I also tried to do something different on the `GstLAL` dataset. So the pipeline recovers $\mathcal{M}_c$ perfectly (I am not even sure that make a regression on $\mathcal{M}_c$ makes sense), then I suppose that it recovers also one of the two masses (I don't know which one) and compute the second one analytically. Therefore the degeneracy problem in one mass should be "specular" in the other masse, so maybe to remove the degeneracy problem in the regression we can consider suitable combinations of the two masses m_i instead of considering them separately. The using this quantity $g(m_1, m_2)$ and $\mathcal{M}_c$ we can obtain two masses m_i. It turns out that symmetric quantities are better, in particular $p_k = (m_1 m_2)^k$ can be regressed very well if the exponent k is not too high, obtaining R^2 coefficients such that $1 - R^2 \leq 10^{-4}$. Then the two masses can be obtained analytically from

$$m_{1,2} = \frac{p^3}{2\mathcal{M}_c^5} \left(1 \pm \sqrt{1 - 4\nu}\right), \tag{9}$$

where $p = m_1 m_2$ (i.e. we omit the index $k = 1$) and ν is the symmetric mass ratio

$$\nu \equiv \frac{\mathcal{M}_c^{10}}{p^5} = \frac{m_1 m_2}{(m_1 + m_2)^2} \in (0, \frac{1}{4}]. \tag{10}$$

However, note that the condition

$$\nu = \mathcal{M}_c^{10}/p^5 \leq \frac{1}{4} \tag{11}$$

is not enforced during the training (i.e. the predictions for p_k and $\mathcal{M}_c$ are not constrained in this sense), then it is possible that (11) will be slightly violated, making the two masses complex. For this reason we recover the two masses with a modified version of Eq. (9)

$$m_{1,2}^{\rm pred} = \frac{p_{\rm pred}^3}{2M_{c,\rm pred}^5} \left(1 \pm \sqrt{1 - 4\min\left(\nu_{\rm pred}, \frac{1}{4}\right)}\right). \tag{12}$$

Note that the condition (11) can be violated pretty often if we have many equal-mass binaries in the dataset (and this is precisely our case). However note that with this approach the predictions for m_i and $\mathcal{M}_c$ can be not consistent, i.e. $\mathcal{M}_c^{\rm pred} \neq (m_1^{\rm pred} m_2^{\rm pred})^{3/5}/(m_1^{\rm pred} + m_2^{\rm pred})^{1/5}$. To enforce this condition we then compute m_2^* requiring

$$\mathcal{M}_c^{\rm pred} \equiv \frac{(m_i^{\rm pred} m_j^*)^{3/5}}{(m_i^{\rm pred} + m_j^*)^{1/5}}, \tag{13}$$

where $i \neq j = 1, 2$, explicitly (omitting the superscript 'pred' in the RHSs):

$$m_j^* = \frac{\mathcal{M}_c^{5/3}\left(3^{1/3} 2 \mathcal{M}_c^{5/3} + 2^{1/3} S_i^2\right)}{6^{2/3} m_i^{3/2} S_i}, \tag{14}$$

$$S_i = \left(9 m_i^{5/2} + \sqrt{81 m_i^5 - 12 \mathcal{M}_c^5}\right)^{1/3}. \tag{15}$$

The results for this kind of regression using the features $p = m_1 m_2$ and $\mathcal{M}_c$ are shown in Fig. 7. We use a NN with one hidden layer with 100 neurons and ReLU, linear activation function in the output layer (i.e. no constraints), MSE loss function and we train for 100 epochs using $N_{\rm batch} = 128$. While the regression on p and $\mathcal{M}_c$ is almost perfect (R^2 coefficients of 0.99993 and 0.99997, respectively), the other recovered quantities show some strange behavior and, most importantly, they are strongly dependent on the initial state on the NN, i.e. training the NN different times using always the same options leads to pretty different results for m_i and q. Moreover, the mean predicted/injected errors for m_1, m_2, $\mathcal{M}_c$, and q are slightly worse since here we get 2.8%, 2.7%, 0.02%, and 5.4% respectively, while with the standard regression of Sec. 5 we got 2.3%, 2.2%, 0.05% and 4.6%.

Another attempt could be to do the regression on ν and $\mathcal{M}_c$ and then compute m_i from these two quantities. In this case we could apply a constrained output function to ν in order to have a more consistent output (and in this case we should write a new class for the linear-scaler since the `MinMaxScaler` that we are using now in the constrained output wouldn't be the best choice for constraining a quantity like ν). However I don't think that this will solve all the problems that we have with this dataset.

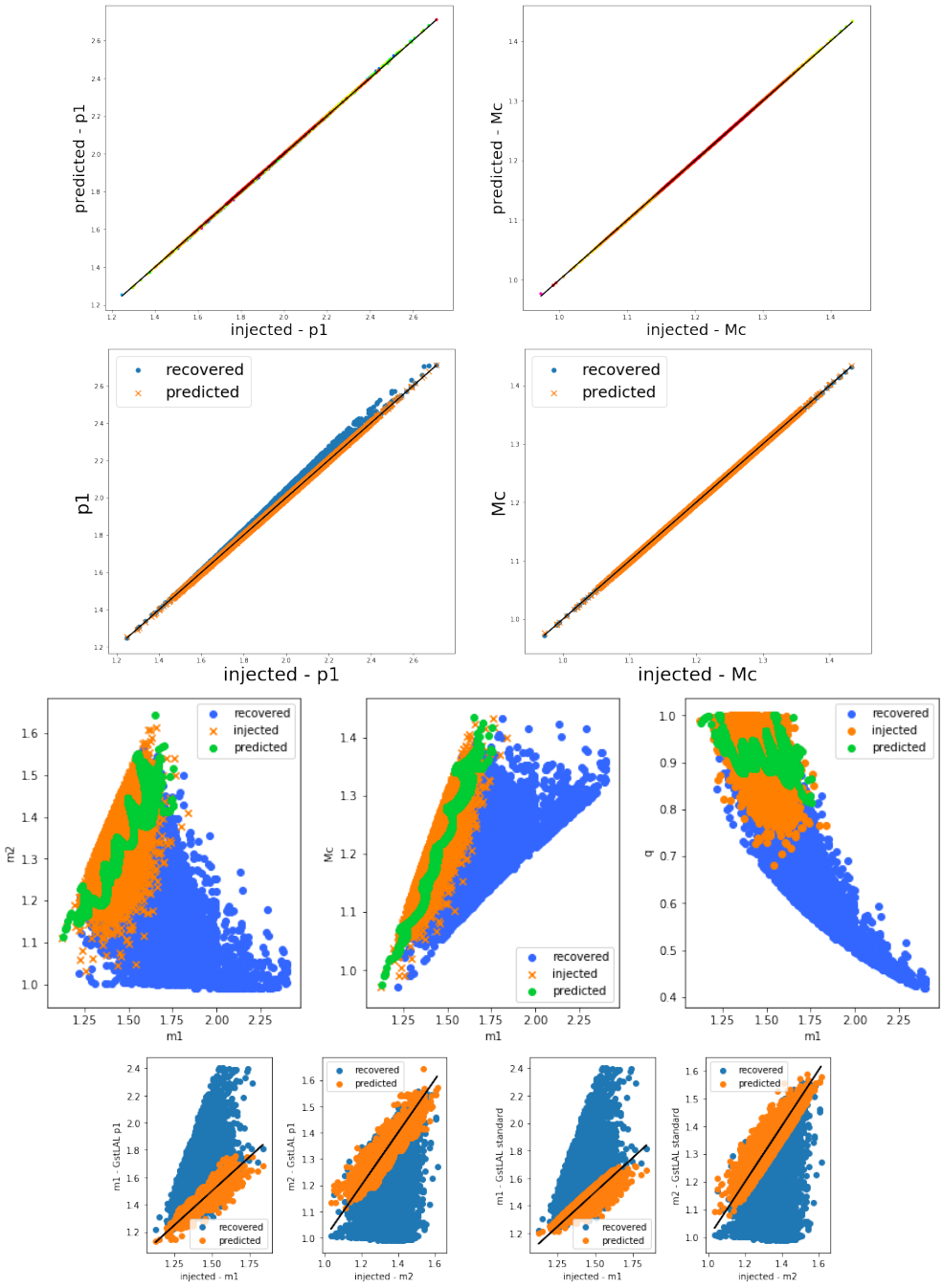

Figure 7: Results of the regression on the `GstLAL` dataset using the approach of Sec. 7. We used one hidden layer with 100 neurons, $N_{\text{batch}} = 128$, 100 epochs, unconstrained output. The colors in the first plots is related to the absolute difference between predicted and injected. The black line is the bisector. The final R^2 coefficients for $p = m_1 m_2$ and $\mathcal{M}_c$ are 0.99993 and 0.99997, respectively. In the three panels in the middle we plot m_2, $\mathcal{M}_c$ and q over m_1 for the injected, recovered and predicted quantities. Finally, in the first two panels of the last row we show the (indirectly) predicted masses with the $(p, \mathcal{M}_c)$ regression versus the injected masses, while in the last two panels of the last row we show the predicted masses obtained with the standard regression of Sec. 5 versus the injected ones.

8 GstLAL BNS dataset - the return of the different approach

Marina thinks (hello) that using different variables constructed from m_1 and m_2 may be a good approach, along the modified loss function to add information without regressing more variables. Also it is true that a NN constructs within new variables that are (non)linear combinations of the input ones, so ideally the NN should do that step for us. Guessing that it does not do so, I propose the following:

Instead of considering just one *made-up* quantity, let's consider two such that

$$p = (m_1 m_2)^3 \qquad s = m_1 + m_2. \tag{16}$$

The idea behind this definition is that the known chirp mass then would be $\mathcal{M}_c = (p/s)^{1/5}$, and we aim to reduce the degeneracy. Once these variables are regressed, the original masses can be recovered by

$$m_1 = \frac{s + \sqrt{s^2 - 4\sqrt[3]{p}}}{2} \qquad m_2 = s - m1. \tag{17}$$

If we analyze the content of the square root we can see that $s^2 - 4\sqrt[3]{p} = (m_1 + m_2)^2 - 4m_1m_2 = (m1 - m_2)^2$ and therefore it would be always positive. This is also why we take its the positive sign. In practice and as the regression is not perfect, the content of the square root can be slightly negative ($\mathcal{O}(10^{-3})$), and then we will take it as zero and both masses would be equal. With this definition there are no conditions of further restrictions.

For the moment, we consider the loss function just *MSE*. Training with 20 epochs, more than enough to saturate the training, we try with different number of hidden layers getting the mean errors that can be seen in figure 8. All the quantities reduce the error with the number of layers but in an oscillatory way. For this study, the minimum error is obtained with 290 layers.

Then we compare results for 80, 100 and 290 hidden layers in figures 9-11. We use these two lower numbers to illustrate how a NN with less parameters behave, and then the optimal one from the nets tried. All obtain a high accuracy in the computed parameters (> 0.99).

Some tests where implemented to control the behaviour of the predicted masses from the parameters s and p proposed: first we control that mass1 is always below the threshold of 3 solar masses to consider it a NS.

Then we can recover mass2 either from the injected chirp mass or from the algebraic relation using s and m_1. So we calculate the difference of the values obtained through these two methods, and show a warning if the difference is bigger than 0.02 (arbitrary reasonably low number). We never get an alert, and some examples of the errors obtained can be seen in figure 10.

Finally in figure 11 we compare the recovered, injected and predicted values for m_2, $\mathcal{M}_c$ and q as a function of m_1. We can see that the predicted values are always over the region of injected values, and not in the region of wrongly recovered ones; I would suggest this is good. But during an interval of m_1 it covers a narrower parameter space that it should. Also it is curious that with 80 and 290 layers we have a very similar behaviour in the plots, but with 100 the results look better. My theory: for $m_1 \in (1.5, 1.75)$ and mass ratio $q < 0.9$ (approximate values from the plots) the regression is more difficult. So when we have less layers the error is bigger, then there is an appropriate number of layers around 100 that performs the best. And finally a very high number of layers overfits the data in the "easier" region getting a better score, but performs equally bad in the complicated region.

9 The loss-function strikes back

The loss function is very important for the NN, but not only its value, but its shape. This is because for updating the parameters in each iteration it does a (kind of) gradient descent, so it is not only the value, but the derivative, what we have to tune. I think that the losses proposed up there are not good because of this.

I propose the loss function

$$J_{\text{mse}} = \text{mean} \sum_i (y_i - \hat{y}_i)^2 \cdot (1 + \lambda(\mathcal{M}_c - \hat{\mathcal{M}}_c)) \cdot \text{SNR} \tag{18}$$

where the hat is the computed value with current parameters and the value without the hat is the

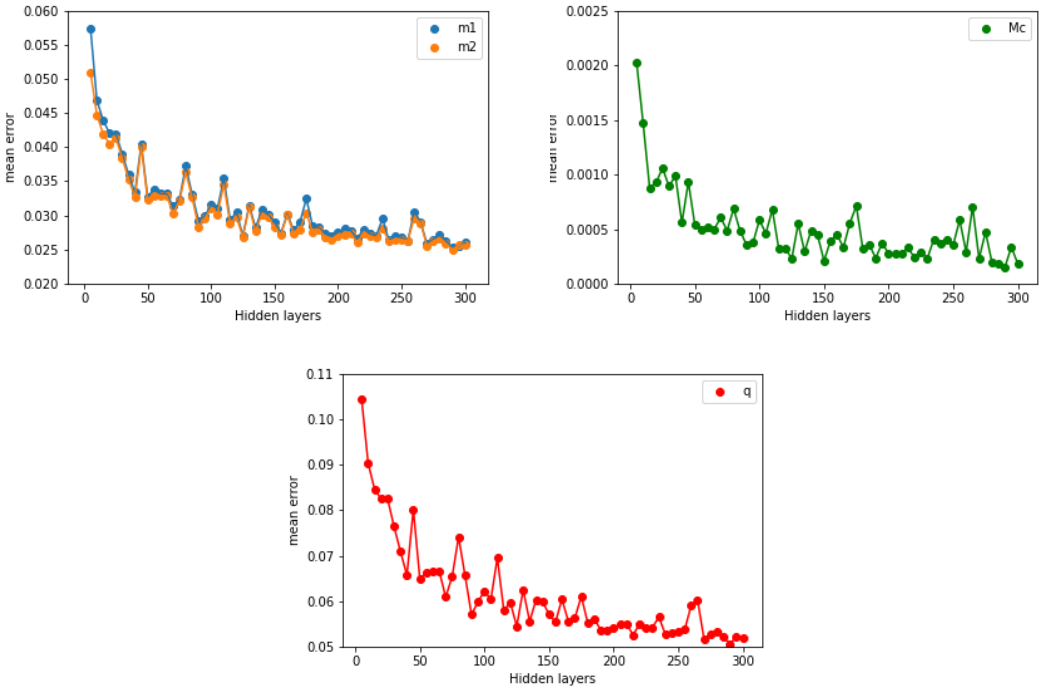

Figure 8: Errors obtained for different hidden layer size using the regression data p and s from Sec. 8

true one. The multiplication by the SNR is just to add information. I guess that a higher SNR would lead to a better reconstructed value of the parameters by GstLAL, and so misleading them would be worse. If this does not make sense, just remove the SNR term.

This shape should work better, because when deriving from respect $\hat{y}$ the penalty is included in the gradient.

10 GstLAL BNS dataset - analysis

The results from regression have the same problems even using different variables. So here I study if the problem is related to the data itself, and no the NN. Here we focus just on the raw data.

In section 8 we get to the conclusion that there is trouble predicting good quantities for $m1 \in (1.5, 1.75)$, $m_2 \in (1.1, 1.4)$ and $q > 0.9$ (approximately). The limits for the problematic region were less clear in previous attempts, e.g. Sec. 7, so this indicates that maybe there is some problem with the data, but also the way of doing the regression is key.

In figure 12 there are the histograms of the data used for training, validating and testing. Although the problematic region of m_2 is full of data, that is not true for m_1 and the mass ratio. In numbers, around 73% of data for every step has $q > 0.9$. And around 76% of the values of mass1 lay out of the signalled interval for all steps.

The result is that the training is done mostly in binaries with $q > 0.9$ such that $m_1 < 1.5$. That is why in that region of the parameter space, and with an optimal configuration of the NN, we can predict values very close to the injected ones. At the rest of the domain, we perform poorly, but best with an intermediate number of layers: not enough layers do not have the power to extrapolate, and too much overfit the data and cannot extrapolate correctly.

This small analysis already indicates the problems of our dataset: it is not equally sampled over all the parameter space. From here we should con-

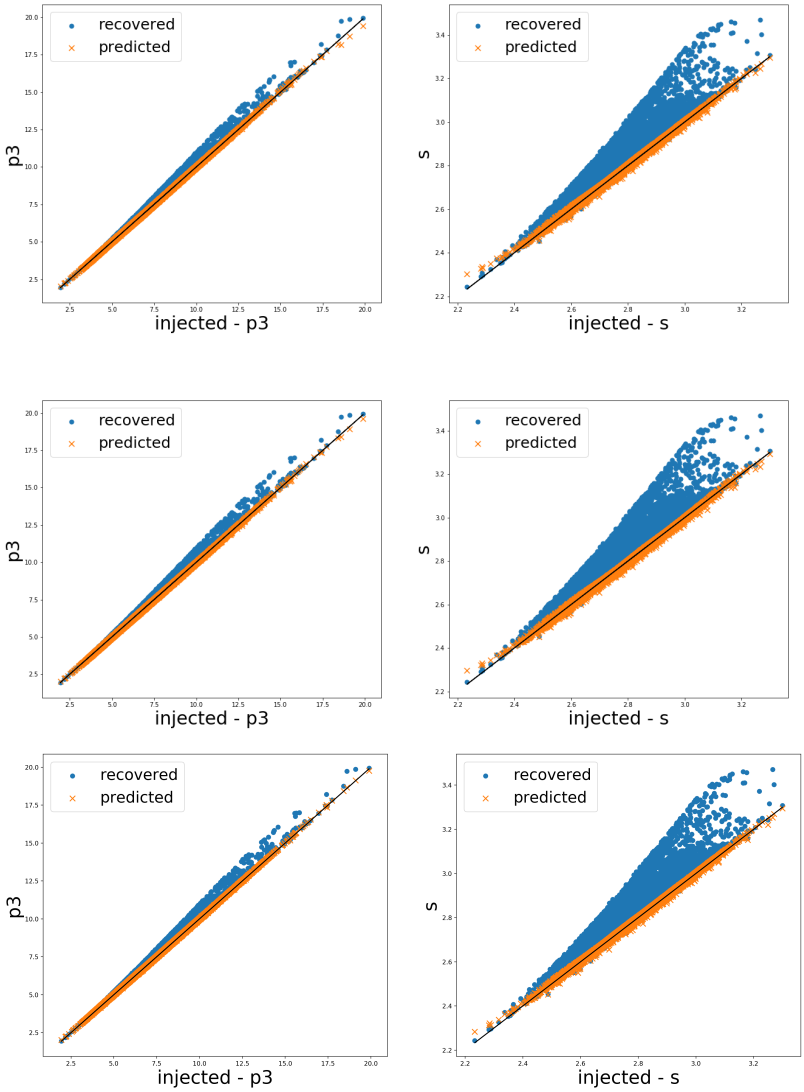

Figure 9: Results from the regression for 80, 100 and 290 hidden layers top to bottom, with approach (s,p).

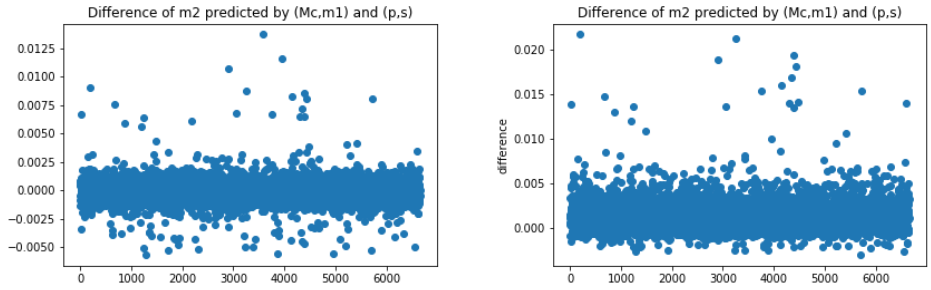

Figure 10: Difference between two ways of obtaining mass2 for 80 hidden layers (left) and 100 (right) with approach (s,p).

sider:

- If with this information we see our results as bad as before, or much better now
- Maybe new metrics that measure the performance related to similar data available for training: like penalizing errors in a populated region, but forgiving bad extrapolations (Is this a thing???)

11 Things to discuss

- The regression with NNs does not work properly on the GstLAL BNS dataset. SOLVED: this is a mix of data not sampled uniformly and bad design of the NN.
- Improve the loss function. We can add information even if it is not a parameter to do regression to. See section 9. Simone can you implement this?
- Does GPR face the same issues? Try with $m1, m2$, and the approach with parameters p and s. If it performs better than the NN, then it is a matter of the loss function for the NN now.
- Improve fake-fake data? Not at the moment.
- Since we are starting to use more complex NNs (at least for `NewRealistic` datasets, i.e. `v0c0` and `v1c0`), should we try to use some dropout? [not a first order priority in any case]

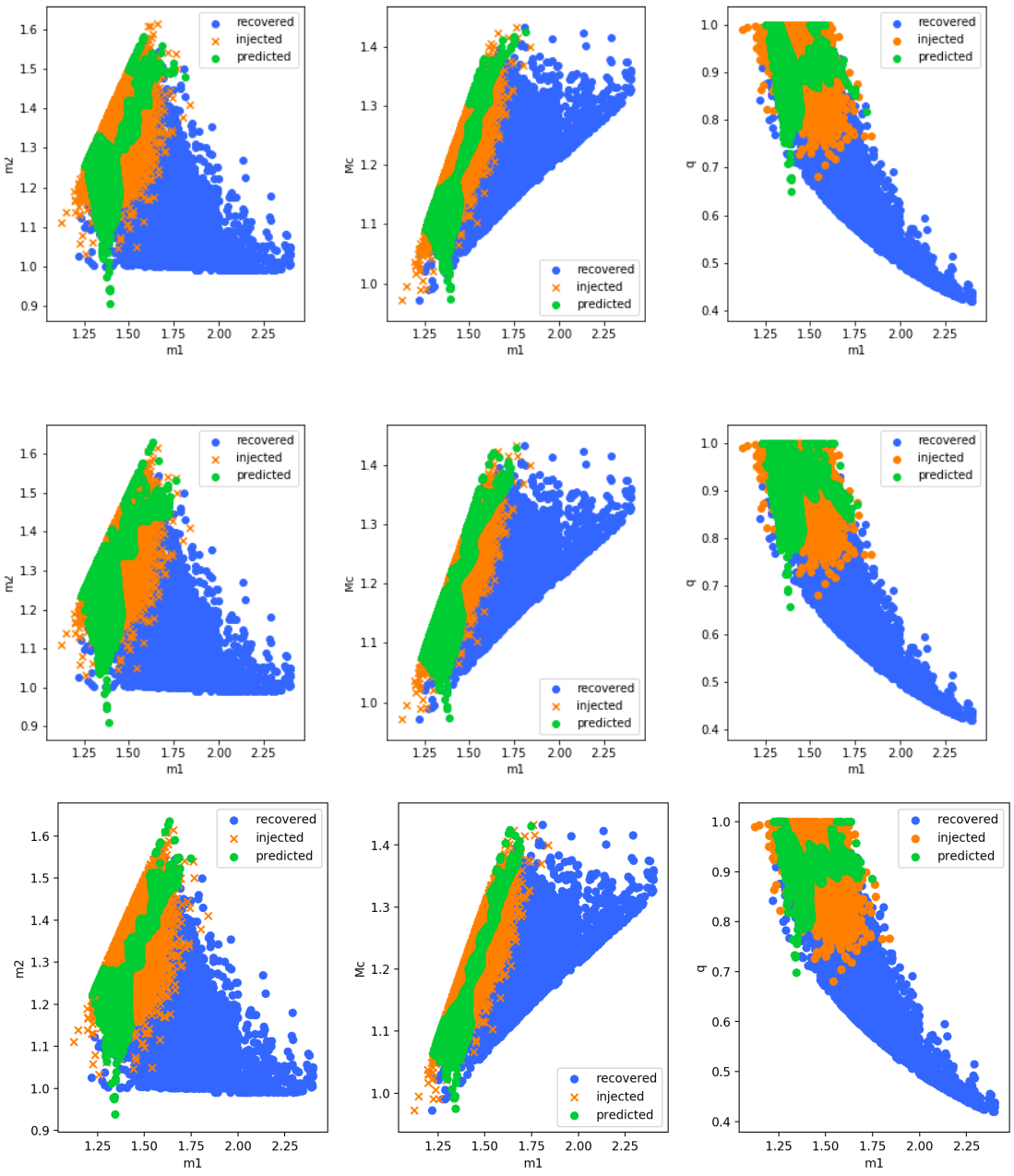

Figure 11: predicted vs recovered vs injected quantities, 80, 100 and 290 hidden layers top to bottom, with approach (s,p).

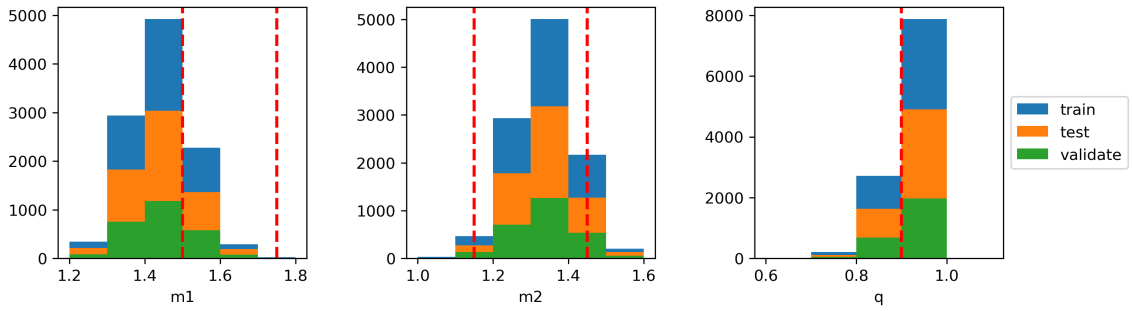

Figure 12: Histograms of the data used for training, validating and testing. For the masses, the red lines enclose the problematic region, and for the mass ratio, we put the division at 0.9.