

Preprocessing, Regularization, and Hyperparameter Tuning

Kathy Breen
March 1, 2019

Things to Consider

- How do we translate real, physical processes to Python?
- What are “dangers” to look out for during the training process?
- Which actions can we take to improve performance, and in what order?

Preprocessing: Why is this important?

- Your data tell a story – are you communicating this to your DLM?
- Smaller numbers take up less space in memory
- Features transformed to similar scales aid in smooth optimization

The most significant improvements in performance are often achieved from changes made during data preparation prior to running a machine-learning algorithm. Neural nets are often referred to as “black box” models, which, while technically correct, the suggestion that the underlying mathematics of the mapping are unknown leads investigators to pay insufficient attention to data preparation.

Data Structure and Feature Engineering

- $M_{\text{samples}} \gg N_{\text{features}}$
- `X.shape = (Nsamples, Nfeatures)`
- Which features are the most salient?
- How to communicate categorical data vs. measured data?

$X =$

	Feature 1	...	Feature N
Sample 1	axis = 0 : operation is performed across all samples for each feature		
Sample 2			
...			
...	axis = 1 : operation is performed across all features for each sample		
Sample M-1			
Sample M			

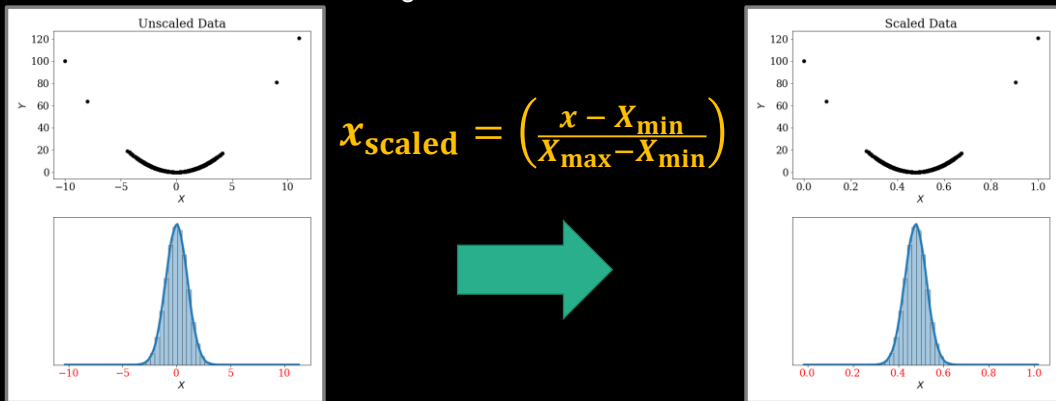
Prepare data such that $N_{\text{samples}} \gg N_{\text{features}}$. If samples are limited ($< 10,000$), only the input parameters to which the prediction of interest are sensitive should be used; however the argument may be made that the introduction of random noise by insensitive features aids in overparameterization and therefore generalization of the trained model [Allen-Zhu et al., 2018].

There is a valid argument for training large, overparameterized networks because they are more likely to be able to generalize a given mapping [Li and Liang, 2018, Allen-Zhu et al., 2018]. One way to overparameterize a network is to create synthetic samples by adding noise to existing samples and concatenating the matrices. This trick has the added benefit of increasing the sample dimension, which may help the network to converge to the "true" solution.

Scaling

- Change the **range** of the data set

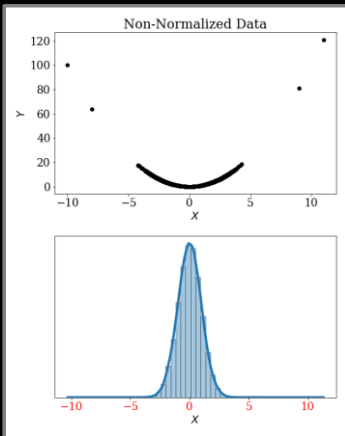
- Helps when comparing data with different units (i.e., different scales)
- Can distort relative ranges between outliers and non-outliers




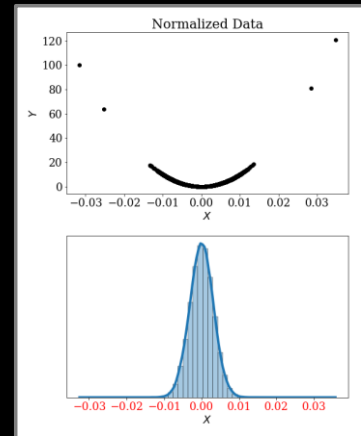
Notice that in the scaled data set, the range between the mean and greatest outliers is skewed. In the original data set, the non-outliers have an absolute range of 10 units, and the outliers were approximately 10 units away from the mean. For the scaled data, the non-outliers now have a much smaller absolute range of approx. 3 units and the outliers are now 5 units away from the mean, i.e. the magnitude of difference between the mean and outliers has been amplified.

Normalization (i.e., Standardization)

- Change the scale of the data set without distorting the relative range



$$\|X\|_2 = \sqrt{\sum_{k=1}^n x_k^2}$$




If your data set has outliers, standardizing the data set is a safer bet than scaling, because the magnitude of the difference between outliers and non-outliers will be preserved. Scaled data with outliers may become skewed. For physical systems, preserving the magnitude of effects is important!

Alters the magnitude (length) of each feature vector to a given norm (L_2)
Aids in stable convergence of weights and biases

Normalized data:

Median = mean

obs. below mean \approx # obs. above mean

```
# create input and output (X, Y) data sets
X = np.random.randn(100000,1)
Y = np.square(X)

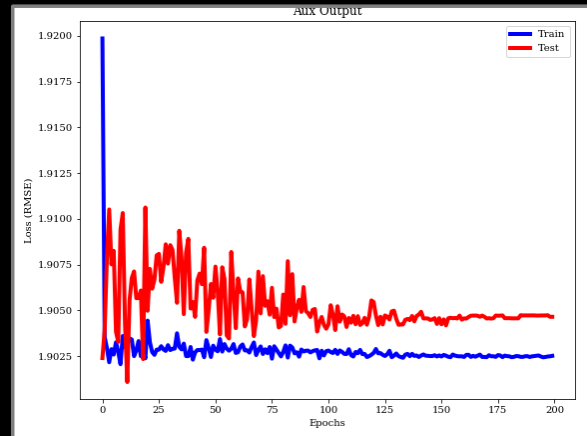
# Preprocess data -----
# normalize inputs for each feature (column) across all samples (rows)
X = preprocessing.normalize(X, norm='l2', axis=0)

# shuffle samples
ismpls = list(i for i in range(0, X.shape[0]))
shuffle(ismpls)
ismpls = np.argsort(ismpls)
X = X[ismpls,:]
Y = Y[ismpls,:]

# partition X and Y into training (90%) and test (10%) sets
split_idx = int(round(X.shape[0]*0.9))
X_train = X[:split_idx,:]
X_test = X[split_idx:,:]
Y_train = Y[:split_idx,:]
Y_test = Y[split_idx:,:]
```

Regularization: Why is this important?

- Prevent model from memorizing patterns in the data as opposed to true learning
- Dropout: stochastic/noise regularization
- Weight-based: penalize large weights that would cause the loss function to “ping pong” around minima



If the number of data points is small compared to the number of model parameters, meaningful regularization has to be employed during the learning process. This can be either stochastic/noise regularization as in the dropout procedure, weight-based regularization by adding discrete penalties for large weights, or by smoothness priors added to the cost function.

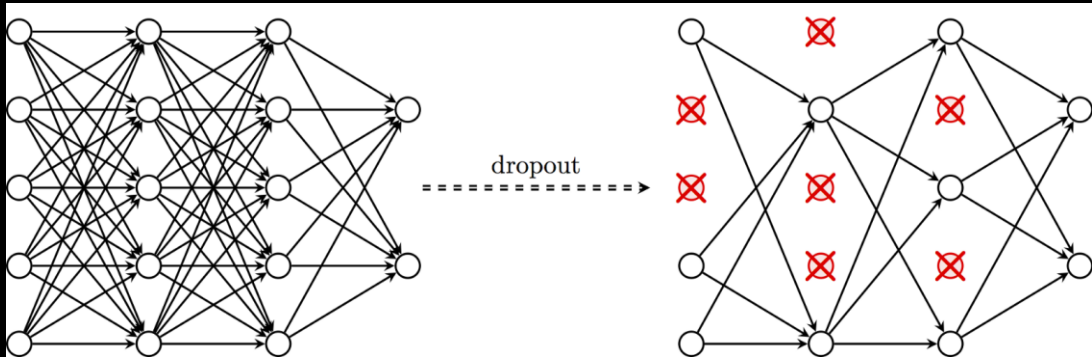
You can include an extra term in the loss function that is formulated as $\alpha ||\mathbf{W}||$. This penalizes high weights in the NN and keeps the magnitudes of weight matrices down. You don't want any specific pathways through the NN to become dominant as this will lead to overfitting. If a few pathways dominate the flow of information through the training data set, they will likely fail on the test data set.

Dropout accomplishes the same thing as \mathbf{W} regularization, but perhaps is a bit of a stronger technique. It randomly drops out the specified percentage of NN connections each pass through the NN. This effectively prevents the NN from becoming too reliant on any few pathways as they could be lost on the next pass through. My experience with dropout is that it almost always improves performance with common values between 0.1 and 0.5 (larger NNs can handle larger dropout). I think you can find some dropout animations if you search for them. And there are

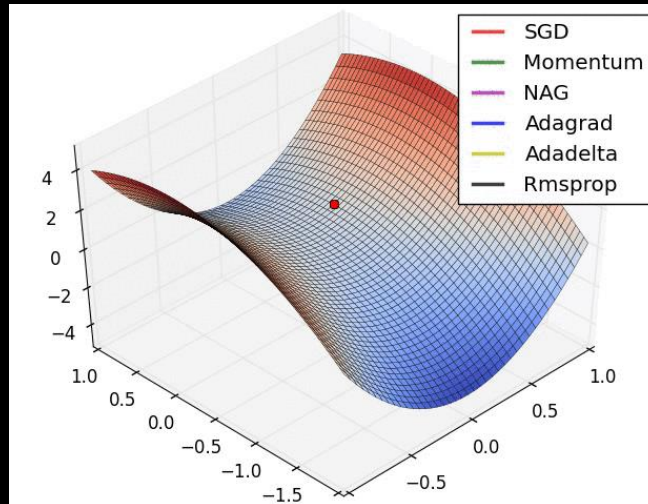
some good YouTube videos that liken it to a game of whack-a-mole. You force the network to learn redundant pathways, which helps prevent overfitting. You whack the mole that is resulting in a large weight (lots of info passing down one pathway).

Batch processing is conceptually easy to understand. When you have a large dataset, you could train it all at once, but forward and backprop are expensive at best, and won't fit into memory at worst. While training on all data at once guarantees that the loss function will always decrease, it is inefficient. Instead, you send just a batch of your data through forward and backprop and update your weight matrices. While the next time you send another batch through forward and backprop you are not guaranteed to see a decreasing cost function, after successive iterations you will see it trend downward. You are essentially updating your weight matrices on only a portion of your data, which does not guarantee convergence each step, but is much faster on the whole.

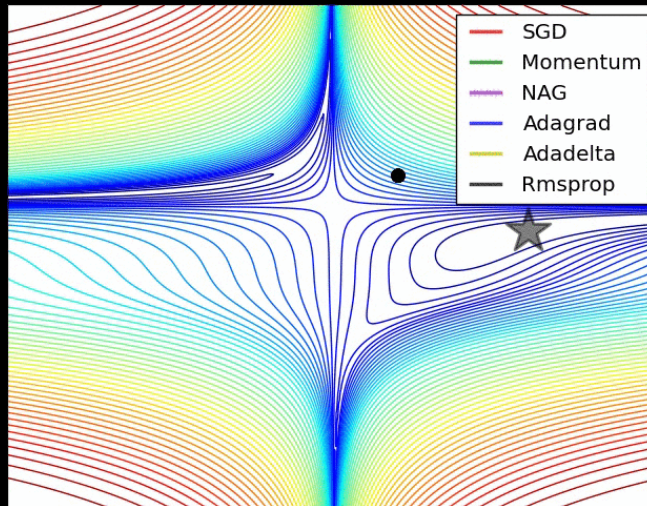
Dropout



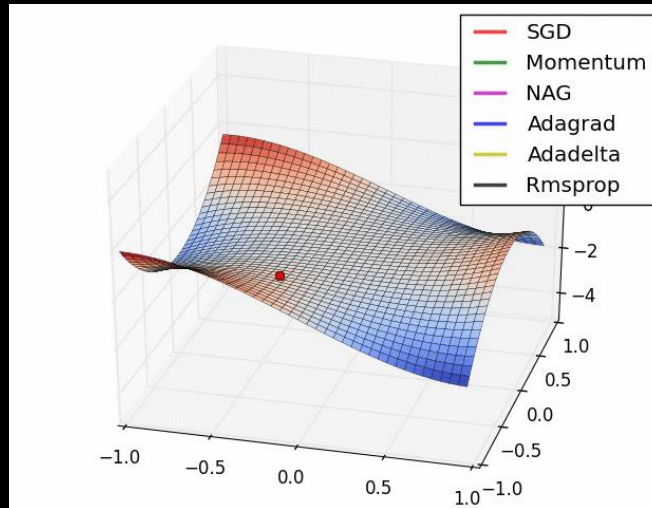
Optimizers: Improve Performance During Gradient Descent



Optimizers: Improve Performance During Gradient Descent



Optimizers: Improve Performance During Gradient Descent



HYPERPARAMETERS

```
epochs = 2000  
batch_size = 5000  
do = 0.2  
N_nodes = 64
```

create input layer.....

```
main_input = Input(shape=(X_Nfeatures),  
                    dtype='float',  
                    batch_shape=(batch_size,X_Nfeatures),  
                    name='main_input'  
                    )
```

create hidden layer.....

```
hidden_layer1 = Dense(N_nodes, activation='relu',  
                      name='hidden_layer1')(main_input)  
Dropout(do)(hidden_layer1)
```

create output layer

```
main_output = Dense(Y_Nfeatures,  
                    name='main_output')(hidden_layer1) # default  
activation is linear
```

feed datasets into model for training

```
model = Model(inputs=[main_input],  
              outputs=[main_output]  
              )
```

compile the model with desired configuration

```
model.compile(optimizer='adam')
```

train the model, and store training information in the history object

```
history = model.fit([X_train],[Y_train],  
                   epochs=epochs,  
                   batch_size = batch_size,  
                   validation_data=(X_test, Y_test) )
```

evaluate the trained model on the test data set

```
X_pred = np.random.randn(100000,1)  
Y_pred = np.square(X_pred)  
predict =  
model.predict([X_pred],batch_size=batch_size)  
Y_mse = mean_squared_error(predict,Y_pred)
```

What are hyperparameters?

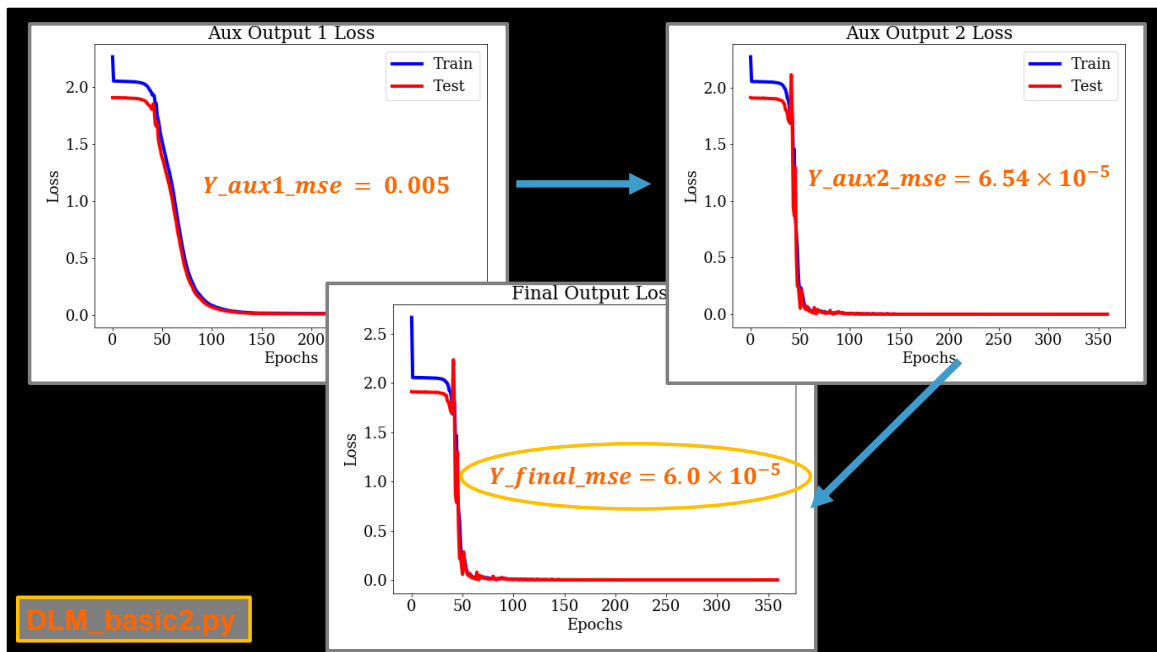
- Machine learning parameters – govern network behavior during the training process to optimize performance
- Ex: Learning rate, dropout rate, number of nodes/layers, optimizers, etc.
- Iteratively train a model with different HP configurations, choose the realization with the best performance (after validation)

Once preprocessing is complete, hyperparameter optimization can provide incremental yet critical performance improvements. Traditionally this is done by randomly choosing configuration settings from the search space and training each configuration individually, then choosing the model with the best performance (once it has been validated). Some argue for more formalized algorithms [Li et al., 2017]).

References

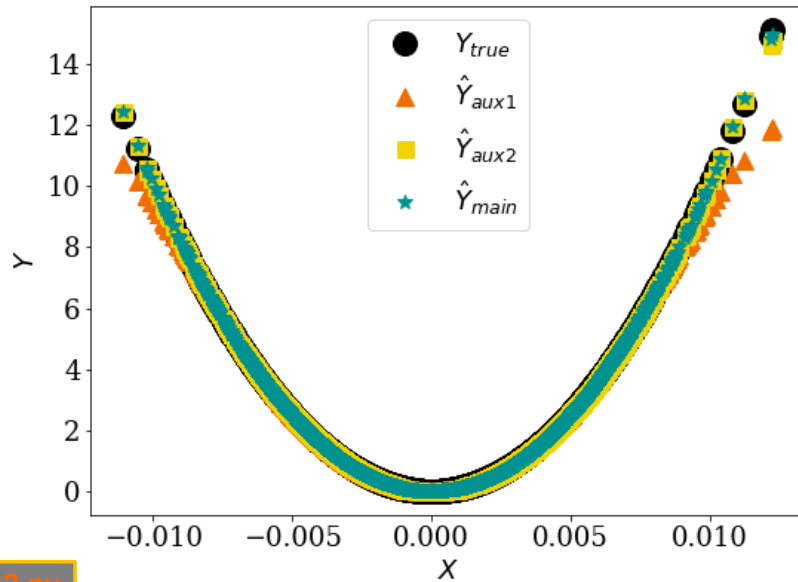
- Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. arXiv preprint arXiv:1811.04918, 2018.
- Koji Hashimoto, Sotaro Sugishita, Akinori Tanaka, and Akio Tomiya. Deep learning and ads/cft. arXiv preprint arXiv:1802.08313, 2018.
- Raban Iten, Tony Metger, Henrik Wilming, Lidia del Rio, and Renato Renner. Discovering physical concepts with neural networks. arXiv preprint arXiv:1807.10300, 2018.
- Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. The Journal of Machine Learning Research, 18(1):6765{6816, 2017.
- Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. arXiv preprint arXiv:1808.01204, 2018.

DEMO



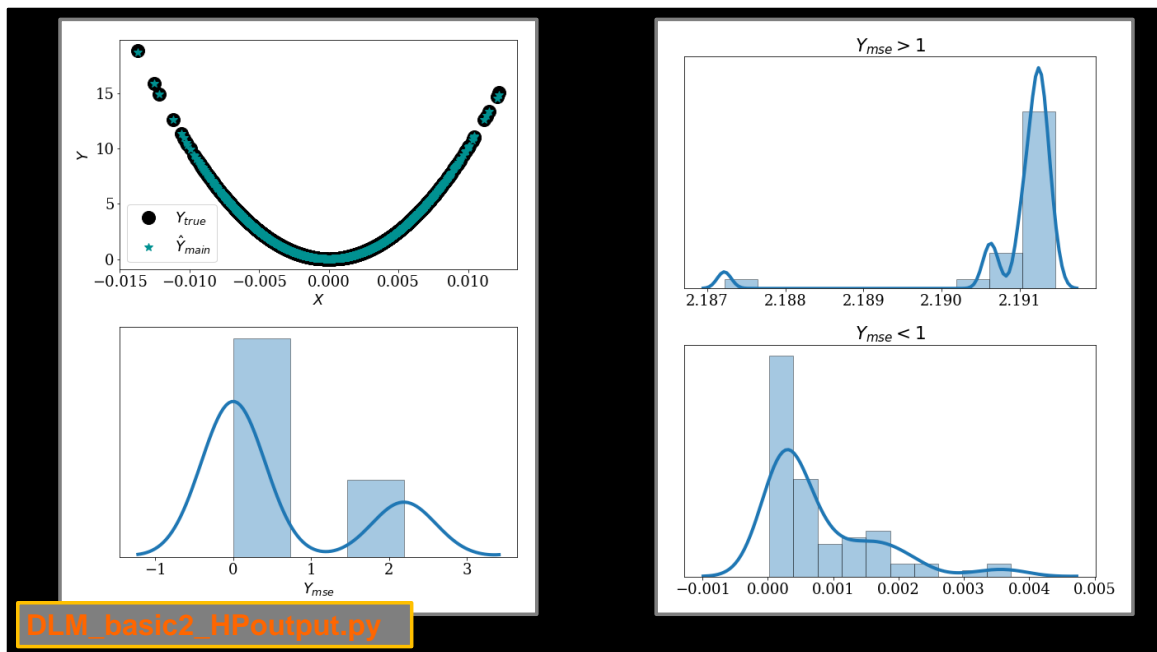
Note that the loss metric (mean squared error) was reduced by two orders of magnitude between the auxiliary loss outputs – this is a significant increase in learning! A smaller, but still significant, reduction in loss is seen between the second auxiliary and final outputs. For a full picture of the model structure, see [DLM_basic2_structure.pdf](#).

Hint: to get an idea about how well/quickly the model is learning between output points, look at the slope of the loss functions – a steeper slope indicates more rapid learning/optimization.



DLM_basic2.py

As indicated in the previous slide, the outputs from the second auxiliary and final loss outputs are very similar, whereas the first auxiliary output does not perform well at the extremes of the function.



Here is a breakdown of the HP tuning runs – most runs fell into one of two categories: “pretty good” and “very wrong”. The best run had excellent predictions – even for extreme values!

For a full picture of the tuned model structure, see `DLM_basic2_structure_tuned.pdf`.