Handling Skewed Target Distribution

I have column in pd df containing normalized values of a target variable. The plot of the value distribution shows a clear power law distribution where lower values are way much over represented in the current sample. Here are 70 random values to give you an idea: [0.001, 0.001, 0.001, 0.001, 0.002, 0.003, 0.005, 0.006, 0.009, 0.009, 0.009, 0.01, 0.011, 0.013, 0.015, 0.016, 0.016, 0.018, 0.021, 0.021, 0.022, 0.025, 0.025, 0.026, 0.026, 0.026, 0.029, 0.03, 0.031, 0.031, 0.033, 0.035, 0.037, 0.038, 0.038, 0.038, 0.039, 0.039, 0.044, 0.057, 0.073, 0.08, 0.083, 0.103, 0.107, 0.107, 0.134, 0.153, 0.2, 0.204, 0.214, 0.233, 0.233, 0.242, 0.248, 0.252, 0.267, 0.272, 0.277, 0.28, 0.282, 0.29, 0.291, 0.308, 0.325, 0.331, 0.381, 0.436, 0.556, 0.585].

I'm trying to train a model to predict such target variable but I'm very concerned about the model having a bias towards the skew direction of the data. I don't have much time to augment less represented instances. Is there any way where I can modify the score values in a way that combats this skew without compromising the integrity of the data? If so please provide the code for such solution. Thanks!

Thanks for the context! To better tailor a solution, could you clarify:

1. What kind of model are you using (e.g., linear regression, tree-based, neural net)?
2. What metric are you optimizing (e.g., MSE, MAE, R^2)?
3. Do you plan to use any transformation on the target during training or prediction (e.g., log transform)?
4. Would it be acceptable to transform the target for training and then inverse-transform it for interpretation?

With this info, I can suggest the most practical and integrity-preserving options for your case.

I'm using an MLP neural network with sigmoid activation at the final layer. I'm optimizing with RMSE. Yes, I don't mind using any transformation on the target (I just want to reduce the skew of the data). Yes, transforming the target for training and then inverse-transforming it for interpretation is fine

Thanks! I'll look into effective transformations you can apply to your skewed target variable—particularly ones suitable for power-law distributions and compatible with your MLP + sigmoid setup using RMSE. I'll prioritize methods that preserve interpretability and work well with inverse-transformation post-training. I'll also include example code for each viable option.

I'll get back to you shortly with the most effective solutions.

Transforming a Skewed Target Distribution for MLP Regression

Introduction

Predicting a heavily skewed, power-law distributed target poses challenges for neural networks. With most target values clustered at the low end and a few extreme high values, a model trained directly on the raw target may become biased. In an RMSE-driven regression, the network could struggle to balance errors: either it **over-focuses on the many small values** (neglecting the rare large ones), or **over-shoots on outliers** (hurting accuracy on the bulk of data). To address this, we can **transform the target variable** to reduce skewness before training. After training, we then **invert the transformation** to get predictions back on the original scale .

In this scenario, the MLP's final layer uses a sigmoid activation (output ∈ (0,1)). This means **target values must be mapped to the 0–1 range** during preprocessing. We'll apply an appropriate transformation (logarithmic, power transform, or quantile normalization) **followed by scaling to [0,1]**. After training, we reverse these steps to recover the original target values. The transformation methods we'll discuss are all monotonic and invertible, ensuring we can interpret predictions in the original units.

Key transformation techniques for long-tailed distributions include:

- **Logarithmic transforms** (e.g. log or log1p)

- **Power transforms** (Box–Cox for positive data, or Yeo–Johnson which handles zeros/negatives)
- **Quantile transform** (mapping data to a uniform or normal distribution)

Each of these can mitigate skewness, but they differ in how they affect the distribution and how easily they integrate with a sigmoid output. We'll explain each method's advantages, limitations (especially in the context of a sigmoid-bounded MLP), and how to implement it in code. Finally, we'll ensure that any transform is paired with an **inverse-transform step**, so predictions can be converted back to the original scale for evaluation and interpretation.

Effects of Target Transformation (Visualization)

Before diving into each method, it's useful to see how these transformations affect a skewed distribution. Below, we simulate a positively skewed target distribution and apply various transforms:

*Effect of different transformations on a heavily skewed target distribution. The **original** histogram (top-left) shows a classic long-tail: most values are low, with a few very high outliers. The **log transform** (top-middle) compresses the range, reducing skew (large values are pulled closer to the bulk, though some skew remains). **Box–Cox** (top-right) and **Yeo–Johnson** (bottom-left) transformations produce more symmetric, bell-shaped distributions by finding an optimal exponent (Box–Cox requires positivity; Yeo–Johnson handles zero/negative values). The **quantile transform to uniform** (bottom-right) flattens the distribution completely – frequent low values are spread out and extreme values are compressed to the upper bound – yielding a uniform [0,1] spread .*

Logarithmic Transformation (Reducing Skew with Log)

Taking logarithms of the target is a simple and effective way to tame a power-law distribution. **Log transforms** (e.g. using natural log or base-10 log) compress large values more than small ones, reducing the disparity between the bulk and the tail. This often makes the distribution more symmetric or at least less extreme. For example, if the target is income or population counts, `log(target)` will pull high-income or large-population values closer to the rest of the data. Indeed, applying an ln transform to skewed data is a common approach to "normalize" it .

Advantages:

- **Simplicity:** Easy to apply and interpret. It's a special case of power transforms (Box–Cox with λ→0).
- **Effective on multiplicative scales:** Turns multiplicative differences into additive ones – a huge outlier becomes a manageable increment on the log scale. This often addresses skewness significantly.
- **Invertible:** Simply exponentiate the prediction to get back to the original scale (if using natural log, use `exp`; for log10, use `10**x`, etc.).

Limitations:

- **Requires positive data:** You cannot take log of zero or negative values. If the target can be zero, a common trick is to use `log(1 + x)` (a.k.a. **log1p**) to handle zero gracefully (since log1p(0) = 0). If negatives exist, you'd need to shift the target upward by adding a constant before log (making it positive) – but this must be accounted for during inversion.
- **Partial skew remains:** Log will **moderate** a heavy tail but might not fully normalize it. The distribution after log can still be skewed, just to a lesser degree (see the histogram above – the log-transformed data still has a tail, but much shorter). More sophisticated transforms (like Box–Cox) can sometimes achieve closer to Gaussian.
- **Output range not [0,1]:** A log transform will yield a range roughly `[log(1+min_y), log(1+max_y)]`. For a power-law, this might be, say, [0, 4] instead of [0, 1000]. We **still need to scale** this to [0,1] for the sigmoid output.

Using log transform in an MLP pipeline: You would apply the log (or log1p) to the target, then scale it to [0,1], train the network, and invert later. Below is an example using NumPy for the log and manual min-max scaling:

```python
import numpy as np

# Example: Suppose y_train is our original target array (shape: [n_samples])
y_train = np.array([...])  # original skewed targets

# 1. Forward transform: log (here log1p to allow zeros)
y_log = np.log1p(y_train)  # apply ln(1 + y)

# 2. Scale to [0, 1] for sigmoid output
y_min, y_max = y_log.min(), y_log.max()
y_train_scaled = (y_log - y_min) / (y_max - y_min)

# ... Now use y_train_scaled to train the MLP with sigmoid output ...
# For example, if using Keras or PyTorch, ensure the network outputs are compared to y_train_scaled.

# After training, suppose y_pred_scaled is the model's prediction on new data:
y_pred_scaled = np.array([...])  # model outputs in [0,1]

# 3. Inversion: scale back to log-domain
y_pred_log = y_pred_scaled * (y_max - y_min) + y_min

# 4. Inversion: apply exponential to get original scale (reverse of log1p)
y_pred_original = np.expm1(y_pred_log)  # exp(y_pred_log) - 1, reversing log1p

print(y_pred_original)  # final predictions in the original target space
```

In this code:

- We used `log1p` so that zero values (if any) map to 0 rather than -∞. If your data has no zeros, `np.log(y)` could be used instead.
- We stored `y_min` and `y_max` from the training set's transformed targets. **It's critical to reuse these same values when inverting** (both for training and any future predictions) . In practice, you'd save these parameters or fit a `MinMaxScaler` object (which can later perform `inverse_transform`).
- After model prediction, we do the inverse scaling and then `expm1` (because we used `log1p`). For a plain log (no +1), you'd use `np.exp`.

Remarks: The sigmoid output will now produce values between 0 and 1 that correspond to the log-scaled target. Because sigmoid can never truly reach 0 or 1, extremely low or high targets may be difficult to predict exactly – but the model can get arbitrarily close (e.g., 0.001 or 0.999). Using log1p ensures the lower bound of `y_log` is 0 (if original min was 0), which nicely aligns with the sigmoid's lower asymptote. The upper bound of `y_log` becomes a finite number (log(1+max)), mapped to sigmoid's 1.0. The network might output 0.99 for the highest values, which after inversion gives nearly the original max. This is generally fine. Just remember to **never train or predict with un-transformed targets** once you adopt this pipeline – always transform first and invert last.

Box–Cox Power Transformation

The Box–Cox transform is a family of power transformations that includes log as a special case. It seeks to **find an optimal exponent λ** to stabilize variance and make the data as normal (Gaussian) as possible. For a target value y, the Box–Cox transform is:

$$y^{(\lambda)} = \begin{cases} \frac{y^{\lambda}-1}{\lambda}, & \lambda \neq 0 \\ \ln(y), & \lambda = 0 \end{cases},$$

where λ is chosen (often via maximum likelihood) from real numbers to best transform the distribution into a normal shape. This can significantly reduce skewness. For example, if λ = 0.5, the transform is essentially $\sqrt{}$y; if λ = 0 (limit), it is log y. The result is usually closer to a bell curve than either the raw data or a simple log.

Advantages:

- **Handles skew flexibly:** By tuning the power λ, Box–Cox often yields a distribution closer to Gaussian than a fixed log transform . It can correct not just skew but also some non-constant variance issues.
- **Statistically grounded:** Many statistical techniques assume normality; Box–Cox is a way to **normalize** data if possible. In the context of neural nets, normal-like targets can be easier to learn.
- **Invertible with known λ:** After prediction, the inverse Box–Cox is defined by:

$$y = \begin{cases} (\lambda \cdot y^{(\lambda)} + 1)^{1/\lambda}, & \lambda \neq 0 \\ \exp(y^{(\lambda)}), & \lambda = 0 \end{cases},$$

where $y^{(\lambda)}$ is the transformed value . This means we can recover the original target precisely (given the λ from training data).

Limitations:

- **Requires positivity:** The Box–Cox formula is only defined for **y > 0**. In fact, scikit-learn's implementation will error if zero or negative values are present . If your target has zeros, you must add a small constant (e.g., +1 or +0.001) to all values before applying Box–Cox. This shift must then be accounted for when inverting. If the target can be negative, Box–Cox is not applicable (use Yeo–Johnson instead).
- **Output can be unbounded:** The transformed values from Box–Cox typically resemble a standard normal distribution (which spans negative and positive). So after transforming, you'll get a range that is not [0,1]. You will need an additional scaling step for the sigmoid.
- **Computational step:** Finding λ is usually done via maximum likelihood estimation. This is fast for a 1D target array, but it's an extra step compared to a simple log. Libraries handle this for you, so it's rarely a concern in practice.

When using Box–Cox in an MLP pipeline, it's convenient to use scikit-learn's `PowerTransformer` (which supports Box–Cox and also handles storing λ and doing inverse transforms). We still combine it with a Min-Max scaler to [0,1] for the sigmoid. Here's how you can do it:

```python
import numpy as np
from sklearn.preprocessing import PowerTransformer, MinMaxScaler

# y_train: original target values, all positive
pt = PowerTransformer(method='box-cox')       # 1. Box-Cox transformer
y_bc = pt.fit_transform(y_train.reshape(-1, 1))  # fit and transform (needs 2D array)

# After Box-Cox, y_bc is roughly Gaussian distributed (mean ~0). It can have negative values.
# 2. Scale Box-Cox output to [0,1]:
scaler = MinMaxScaler(feature_range=(0, 1))
y_train_scaled = scaler.fit_transform(y_bc)  # now scaled to 0-1
```

```
# Train the MLP on X_train with y_train_scaled as target...

# After training, invert on predictions:
y_pred_scaled = np.array([...])              # model's sigmoid outputs on test set
y_pred_bc = scaler.inverse_transform(y_pred_scaled)   # back to Box-Cox domain
y_pred_orig = pt.inverse_transform(y_pred_bc)         # back to original target
domain

print(y_pred_orig.flatten())  # final predictions in original scale
```

A few things to note:

- We reshaped `y_train` to (-1,1) because scikit-learn transformers expect a 2D array of shape `(n_samples, n_features)`. Here the "feature" is just the target itself. After transformation, we flatten it back or keep it 2D consistently.
- The `PowerTransformer` with `method='box-cox'` finds the optimal λ and applies the transform. We don't have to manually compute λ or the formula. The fitted transformer stores λ internally and knows how to invert the transform (`pt.inverse_transform`).
- We used `MinMaxScaler` to scale the Box–Cox output to [0,1]. Notice we fit the scaler *after* the Box–Cox transform. (Alternatively, one could integrate this in a single pipeline or use `TransformedTargetRegressor` – discussed later – to streamline the process.)
- Finally, we invert by first doing `scaler.inverse_transform` (bringing data from [0,1] back to the "Box–Cox space"), then `pt.inverse_transform` (applying the Box–Cox inverse formula with the stored λ) to get the original values. The result `y_pred_orig` should align with the original target's scale and distribution.

Box–Cox is particularly useful if your target is strictly positive and you suspect a non-linear power relationship. It will **automatically decide** if, say, $\sqrt{y}$ ($\lambda=0.5$) or $\log y$ ($\lambda\approx 0$) or maybe y^2 ($\lambda=2$) best normalizes the data. In practice, λ is often between 0 and 1 for right-skewed data. By making the target distribution more symmetric, the MLP might learn better (it doesn't have to accommodate extremely high variance in one end of the target range). Just remember that after training, you must undo both the scaling and the power transform to get meaningful predictions.

Yeo–Johnson Power Transformation

The Yeo–Johnson transform is a **power transform similar to Box–Cox, but it relaxes the requirement of positivity** . This means Yeo–Johnson can be applied to data that include zero or negative values. It defines a separate formula for negative values so that the transform is continuous and invertible across the whole real line. In effect, Yeo–Johnson extends Box–Cox to handle zero/negative by shifting and reflecting those values internally (without you manually adding a constant).

Like Box–Cox, Yeo–Johnson has a parameter λ that is typically chosen to maximize normality. When λ=0, Yeo–Johnson defaults to a log-like transform (with adjustments for sign), and when λ=2, it is close to no transform. The transformed data is usually approximately Gaussian.

Advantages:

- **Works with zero or negative targets:** No manual offset required. If your target can be negative (e.g., profit/loss values, temperature changes, etc.), Yeo–Johnson is appropriate whereas Box–Cox isn't . It will find a power transformation for positive values and a separate form for negatives automatically.
- **Flexible like Box–Cox:** Since it optimizes λ, it can effectively reduce skew and normalize the distribution (almost as well as Box–Cox does for positive data). If the data is all positive, Yeo–Johnson's result will be very similar to Box–Cox's result for the same λ.
- **Invertible:** The transform is one-to-one. Libraries handle the inverse for you given the fitted λ (scikit-learn's `PowerTransformer(method='yeo-johnson')` has `inverse_transform` as well).

Limitations:

- **Range after transform:** The output of Yeo–Johnson, like Box–Cox, is unbounded (typically ends up Gaussian-like, spanning (−∞, +∞)). So again we need a scaling to [0,1] for the network.
- **Not magic for extreme outliers:** If the distribution is extremely heavy-tailed, Yeo–Johnson will mitigate it but might not completely flatten it. It aims for normality, which still might leave some probability in the tails (just as a normal distribution has tails).
- **Computational considerations:** Yeo–Johnson uses an iterative procedure to find the best λ (but this is usually fast for 1D targets, and done once during fitting).

Using Yeo–Johnson in practice: It slots in almost identically to the Box–Cox code path, except we specify `method='yeo-johnson'`. For example:

```python
from sklearn.preprocessing import PowerTransformer, MinMaxScaler

# y_train: original target values (can be positive, zero, or negative)
pt_yj = PowerTransformer(method='yeo-johnson')   # 1. Yeo-Johnson transformer
y_yj = pt_yj.fit_transform(y_train.reshape(-1, 1))   # transform data

# 2. Scale to [0,1] for network
scaler = MinMaxScaler((0, 1))
y_train_scaled = scaler.fit_transform(y_yj)

# ... train neural network on y_train_scaled ...

# Inversion on predictions:
y_pred_scaled = np.array([...])                       # model output
y_pred_yj = scaler.inverse_transform(y_pred_scaled) # back to YJ space
y_pred_orig = pt_yj.inverse_transform(y_pred_yj)    # back to original space
```

This is almost identical to the Box–Cox example— `PowerTransformer` simply applies Yeo–Johnson instead. One difference is that **Yeo–Johnson can handle zeros**, so we didn't need to worry about adding a constant. For instance, if `y_train` contained 0 or negative values, `pt_yj` would still work (internally, it uses a formula for y<0). The inversion step will correctly map the network's outputs back, yielding possibly negative predictions if that's appropriate.

In summary, use Yeo–Johnson when your target distribution is skewed but cannot be log-transformed due to non-positive values. It gives you the benefits of a power transform without manual data massaging. The resultant distribution is easier for the MLP to model, and you can still recover original values after predictions.

(Technical note: If your data is strictly positive, Box–Cox and Yeo–Johnson often yield very similar λ and results. You could use either. Box–Cox might have a slight edge in well-established theory for positive data, but in practice the difference is minor. Many practitioners just use Yeo–Johnson for convenience on any data since it auto-handles zeros.)

Quantile Transformation (Mapping to Uniform or Normal)

Quantile transformation takes a completely different approach: **it uses the empirical cumulative distribution function (CDF) to remap values to a new distribution**. In simpler terms, it converts each target value into a percentile (rank) and then converts that percentile into a value from a target distribution (uniform or normal typically). This can **eliminate skew entirely** because it doesn't assume any parametric form – it just spreads data out according to their ranks.

Two common variants are:

- **Quantile to Uniform:** Map data percentiles to [0,1] uniformly. After transformation, the values are approximately uniformly distributed between 0 and 1 (each quantile bin has equal representation).
- **Quantile to Normal:** Map data percentiles to a standard normal distribution (mean 0, std 1). After transformation, the values follow an *almost* Gaussian distribution (i.e., the sorted values align with the Gaussian quantiles).

For an MLP with sigmoid output, the **uniform output is very convenient**: it naturally results in targets in the 0–1 range, so we may not even need an extra MinMax scaling step. We can transform the target to uniform [0,1], train the model, and then use the inverse transform to get back original values. (If we used the normal option, we would then have to scale that normal distribution into 0–1 or use a different output activation that covers ℝ.)

Advantages:

- **Completely removes skew:** By definition, the transformed data have a *flat* distribution (if uniform) or a symmetric bell curve (if normal). The heavy tail is effectively tamed because those few high values all get mapped to high percentiles near 1.0, but not *crazy* outliers. This greatly **reduces the impact of outliers**, as any value beyond the training 99th percentile will just map to something close to 1.0. The many low values will be spread out between 0 and some intermediate quantile instead of all piled at the bottom.
- **No parameter assumptions:** Unlike log or Box–Cox which assume a certain functional form, quantile transform is non-parametric. It will **always use the data's own distribution** to do the mapping. This makes it very robust – even bizarre distributions can be made uniform.

- **Naturally bounded [0,1] (for uniform output):** If we choose uniform output, after the transform every target value lies between 0 and 1. This directly matches the sigmoid range. In fact, the smallest original value will map to ~0.0 and the largest to ~1.0. That means we can feed these to the network without additional scaling. (Do ensure to treat these as regression targets still, not as class probabilities – they're continuous values in [0,1].)
- **Invertible (with interpolation):** QuantileTransformer in scikit-learn can invert the transform using the stored quantiles. Essentially, it will take a predicted quantile (0–1) and map it back to the corresponding value in the original distribution. This works as long as the prediction is within [0,1]. Any slight overshoot (like 1.02) would be clipped to 1.0 by the transformer.

Limitations:

- **Distorts relationships:** A quantile transform is a **highly non-linear, monotonic mapping**. It preserves the rank order of targets but *not* the relative spacing. This can distort the relationship between features and target. For example, if the true relationship was exponential, a log transform makes it linear (easier for a linear model), whereas a quantile transform will scramble the actual functional form into whatever yields a uniform output. An MLP can still learn it since it's non-linear, but we lose any simple interpretable link. In other words, the mapping **may reduce correlation with inputs**, making the learning solely reliant on rank information. This isn't necessarily bad, but it's something to be aware of . Often it's a trade-off: you remove skew (good for uniform coverage) at the cost of introducing a more complex mapping.
- **Clipping at bounds:** By design, any values beyond the training data's min or max **will be clipped to 0 or 1** after transformation . That means if in deployment you encounter a target smaller than anything seen before, it will map to 0 (and similarly large ones to 1). The network then can only predict 0 or 1 for anything beyond the seen range. Essentially, the model cannot extrapolate beyond the observed target range – it will saturate. If extrapolation is needed, quantile transform might not be appropriate. (However, with power-law targets, extrapolation is always tricky; quantile at least bounds the uncertainty.)
- **Data size:** QuantileTransformer in sklearn by default uses up to 1000 quantiles. It needs enough data to estimate the CDF. If you have very limited data, the empirical CDF might be coarse. But in most scenarios, this isn't a big issue; with large data, it's fine.

- **Not one-to-one if duplicates:** If many target values are exactly identical (e.g., many zeros), their quantile ranks will be the same, and after transform they’ll all collapse to the same value. This isn’t a problem in regression per se (the model can still predict that value), but it means the inverse transform of a range in (0,1) might all map to that same original value range. Generally, if your target has ties, any monotonic transform has this issue.

Using quantile transform in our pipeline is straightforward with scikit-learn’s `QuantileTransformer`. We will choose `output_distribution='uniform'` to get a 0–1 output. (We also set a random_state for reproducibility since the algorithm might sample for performance.) For example:

```python
from sklearn.preprocessing import QuantileTransformer

qt = QuantileTransformer(output_distribution='uniform', n_quantiles=1000, random_state=0)
y_train_uniform = qt.fit_transform(y_train.reshape(-1, 1))

# y_train_uniform is now in [0,1], roughly uniform.
# We can train the MLP with y_train_uniform as the target.

# After training:
y_pred_uniform = np.array([...]).reshape(-1, 1)   # model predictions, should be in [0,1]
y_pred_orig = qt.inverse_transform(y_pred_uniform)
```

A few details:

- We specified `n_quantiles=1000` (the default is 1000 or `len(data)` if smaller). This determines the resolution of the quantile mapping. If you have fewer than 1000 points, it will effectively use each point as a quantile. If you have more, it uses 1000 evenly spaced quantiles for efficiency. Generally, the default is fine.
- After `fit_transform`, `y_train_uniform` contains values between 0 and 1. The smallest target becomes 0.0, the largest becomes 1.0, and the rest are spread out. Note that if your largest value was an extreme outlier far above the second-largest, both will still just map near 1.0 (they get bunched at the top). This is actually desirable to reduce that outlier’s influence on training .

- We didn't explicitly use a `MinMaxScaler` here, because the quantile transform already produces values in [0,1]. (If we had chosen `output_distribution='normal'`, we'd get roughly N(0,1) values and would then need to scale or adjust the network output range differently, e.g., using a linear output neuron instead of sigmoid. But since we use uniform, sigmoid is perfect.)
- Inversion is done by `qt.inverse_transform`. We reshape `y_pred_uniform` to (-1,1) as required. The inverse transform will map each predicted quantile back to an original value. For instance, if the model outputs 0.95, `inverse_transform` will give approximately the 95th percentile value from the original training targets.

Important: Because the network is trained on uniform targets, it's essentially learning to predict the **quantile** of the target distribution given the inputs. If your model predicts 0.5, it's saying "the input likely corresponds to the median of the original target." Predicting 0.99 means "this input corresponds to a value near the max of what was seen (or beyond)." This is a valid approach, but keep in mind the model is not directly minimizing MSE in original space, it's minimizing it in quantile space (thanks to our RMSE on the transformed target). This usually still correlates with doing well on original RMSE, especially if the transform is monotonic. Just ensure you evaluate error after inverting back to original scale.

Comparison of methods: In practice, you might try a few transformations and see which yields the best validation performance. Log is simple and often effective for mild skew. Box–Cox/Yeo–Johnson can further normalize distributions and often work well if the log didn't fully remove skew. Quantile transform is very powerful in equalizing the distribution but be cautious about the potential distortion of the target relationship . It's a bit of a black-box transform since it blindly forces uniformity. Some practitioners prefer not to use quantile transform unless necessary, because it can be harder to interpret the model's behavior afterward. That said, it can sometimes dramatically improve training on highly non-linear, imbalanced targets.

Mapping Targets to (0,1) for Sigmoid Output

Regardless of which transformation you use, a final **min-max scaling step to [0,1]** is needed (except when using quantile to uniform, which already yields [0,1]). The sigmoid activation constrains predictions to (0,1), so we must ensure the transformed target falls in that range to avoid clipping. Here are a few guidelines for this step:

- **Fit scaling on the training target only:** Compute the min and max (or the quantile transform, etc.) using only the training data. Then apply the same parameters to validation/test sets. This avoids label leakage. For example, `MinMaxScaler` or `QuantileTransformer` should be `.fit` on y_train, then use `.transform` on y_val and y_test.
- **Avoid extreme compression:** If your target has a couple of *very* extreme outliers, a normal min-max scaling will squeeze most values into a tiny range (because the max is so high). This might hurt training, as most targets will be nearly 0 after scaling. One strategy (as one data scientist suggests) is to scale between the minimum and the 95th percentile (instead of the true max) . For instance, map min to 0 and the 95th percentile to 1.0; any values above the 95th percentile are capped at 1. This ensures the bulk of data is well-spread in 0–1. The downside is that you're no longer strictly invertible for the top 5% – all those outliers beyond P95 are treated as "1.0" by the model. If exact invertibility of every point is required, it's better to include the max in scaling (or perhaps remove/handle outliers separately before modeling). If you do use a percentile cap, be aware that the model cannot distinguish values above that cap (it will predict at most the capped value on invert).
- **Sigmoid range handling:** As noted, sigmoid never outputs exactly 0 or 1 for finite inputs – but it can get extremely close. If your transformed target includes exact 0 or 1, it's not a big issue; the network can still approximate them. You might find it easier to avoid true 0/1 in training targets (maybe use 1e-6 and 0.999999 as bounds) so that the gradients don't go to zero, but this is usually only a concern in classification with cross-entropy. With RMSE loss on a regression target, predicting 0 when the target is 0 is fine (the error is 0). The network will adjust weights to try and match the targets. So, it's acceptable to have 0 and 1 in the scaled target as long as they came from finite original values. Just ensure your model is properly learning a regression (MSE loss on continuous output) and not treating it as a classification problem.
- **Store parameters for inversion:** If you manually compute min/max or use a `MinMaxScaler`, save those values or the fitted scaler. The same goes for λ in Box–Cox or the fitted quantile transformer. This is crucial for the **reverse mapping of predictions** . Using scikit-learn transformers helps because you can serialize the transformer or keep it in memory for inversion. If coding manually, document the values. (In a production pipeline, you might integrate these in a single `Pipeline` or use `TransformedTargetRegressor` to tie it with the model.)

Below is a brief illustration of using `TransformedTargetRegressor`, a scikit-learn utility that wraps a regression model and automatically applies an inverse transform on predictions. This example uses a pipeline: log transform + min-max scaling as the target transformer:

```python
from sklearn.compose import TransformedTargetRegressor
from sklearn.neural_network import MLPRegressor

# Define the transformation for the target:
# Here we use a function transformer for log1p and its inverse (expm1),
# and we'll manually handle scaling in those functions for demonstration.
import numpy as np
func = lambda y: np.log1p(y)     # forward transform
inv_func = lambda y: np.expm1(y)  # inverse transform

# We can combine scaling by extending these functions,
# but for simplicity, let's use sklearn's built-in transformers via a pipeline.

from sklearn.preprocessing import FunctionTransformer
log_transformer = FunctionTransformer(func=func, inverse_func=inv_func)

# Now create the regressor with target transformation:
model = TransformedTargetRegressor(
    regressor=MLPRegressor(hidden_layer_sizes=(100,100), activation='logistic', max_iter=200),
    transformer=log_transformer
)
# Note: scikit-learn's MLPRegressor with activation='logistic' gives a sigmoid on each neuron,
# but for regression it's better to use 'identity' (no activation) in the output layer.
# sklearn doesn't expose output activation easily, so this is just illustrative.
# One might use a pipeline where the final step is a custom sigmoid layer or use another library.

# Train the model (this will internally transform y_train using log):
model.fit(X_train, y_train)

# Predict (the model internally inverses the transform on output):
y_pred = model.predict(X_test)
```

In practice, for an MLP, you might use Keras or PyTorch for more control over the architecture (e.g., a single sigmoid output neuron). The concept remains: apply transformations to `y_train` before computing loss, and invert the predictions afterward.

Conclusion

Choosing a transformation: If your target has a **mild skew** and is strictly positive, a simple **log transform** may suffice to reduce bias towards the lower end. If the skew is stronger or you want the data to be more Gaussian, consider **Box–Cox** (if positive) or **Yeo–Johnson** (if zeros/negatives present) to automatically find a better exponent – these will make the distribution more symmetric, which can stabilize learning. In cases of **extreme skew or many outliers**, a **quantile transform** to uniform can be a game-changer by equalizing the target distribution (just be mindful of the trade-offs in interpretability and extrapolation).

All these transformations are **invertible**, allowing you to train the model in a space where the data are well-behaved, and then translate the outputs back to the original scale for evaluation . The final layer’s sigmoid necessitates that we constrain targets to (0,1) during training, which we achieve via normalization steps. Always test that your inverse transform works correctly by taking a few sample predictions through the full reverse pipeline to ensure you recover reasonable original values.

By applying these techniques, your MLP can learn without being overwhelmed by the imbalance in the target distribution. You’ll likely observe faster convergence and better accuracy, especially on those high-value outliers, once the target is transformed to a more training-friendly shape. Remember to measure error on the **original scale** (after inverting) to confirm that the transformation indeed improved the real-world performance of the model. Each method has its pros and cons, so it’s worth experimenting to see which yields the best results for your specific dataset and task. Good luck with your modeling!

Sources: Transforming skewed regression targets and inverting the predictions

Sources

Explain to me how is the lambda produced in the Box-Cox produced. I'm just curious make your explanation easy to grasp and concise

Box-Cox transformation finds the best **λ (lambda)** by maximizing how close the transformed data is to a **normal distribution**. Here's the simplified, intuitive explanation:

1. **Start with your data**:
   - The Box-Cox transform formula is:

$$y^{(\lambda)} = \begin{cases} \frac{y^{\lambda}-1}{\lambda}, & \lambda \neq 0 \\ \ln(y), & \lambda = 0 \end{cases}$$

2. **Test different λ values**:
   - Box-Cox checks many possible λ values (e.g., from -5 to +5).
   - For each λ, the transform above is applied, and the resulting distribution is tested for how "Gaussian-like" (normal) it is.
3. **Select the best λ**:
   - The chosen λ is the one that maximizes a statistical measure called the **log-likelihood**, which essentially quantifies how close the transformed data is to a normal distribution.
   - In other words, Box-Cox chooses the λ that makes your transformed data as symmetrical and bell-shaped as possible.

Intuition behind λ:

- If **λ ≈ 0**, your data is strongly skewed, and Box-Cox essentially chooses a log transform (`ln(y)`).
- If **λ ≈ 0.5**, it means a square root transform (`√y`) fits best.
- If **λ ≈ 1**, no transform (linear) is necessary, as your data is already roughly normal.
- Negative λ values mean your data requires a reciprocal transformation (`1/y`), indicating strong skew in the opposite direction.

Example:

- **λ = 0** → Log transform
- **λ = 0.5** → Square root transform

- **λ = 1** → No change
- **λ = -1** → Reciprocal transform (1/y)

In short, Box-Cox finds the best λ by automatically checking which value of λ produces the **most normal** transformed data.