Balanced Regression NN Model

I want to use a regression neural network model for the prediction of 2 continuous variables. How do I go about ensuring a balanced data distribution for such task? And I want to you to suggest state of the art neural architecture/libraries for the model, given that the input is going to be of around 700-2000 dimensions.

To help you better with ensuring a balanced data distribution and suggesting the most suitable state-of-the-art regression neural network architecture, could you please clarify a few things:

1. What are the two continuous target variables you're trying to predict (e.g., temperature and humidity, or sales and profit)?
2. Can you describe the nature of the input features (e.g., are they sensor readings, text embeddings, image features, etc.)?
3. What is the size of your dataset (number of samples)?
4. Are both target variables equally important, and do they have similar or different scales?
5. Are there any particular latency, computational, or deployment constraints (e.g., needs to run on edge devices, or inference must be under 100ms)?

Once I have this, I can dive into the best techniques and architectures tailored to your use case.

the continuous target variables synthesized variables generated via an internal engine of my company. The input features are a combination of both text and image embedding data generated using a text and image transformer encoders. I just want to create a proof of concept for now, so my number of samples is on the lower side. The target variables are independent and equally important but both are supposed to be averaged over fragments of variables (I need your help to create a potent averaging and normalization methods based on the distribution and so on). No constraint of deployment, just need accurate results and an architecture suitable for the relatively small sample number for the proof of concept.

Thanks! I'll look into effective techniques for handling imbalanced or irregular data distributions for regression tasks, especially when predicting two independent continuous variables. I'll also find suitable neural network architectures and libraries that perform well with high-dimensional multimodal embeddings (text and image) and relatively small datasets.

I'll include suggestions for:

- Data normalization and averaging techniques tailored to your target variable distribution
- Data augmentation techniques for small datasets with multimodal embeddings
- Regularization and optimization strategies
- SOTA neural architectures and frameworks suited for this use case

I'll get back to you shortly with the results.

Multimodal Regression with Limited Data: State-of-the-Art and Best Practices

Introduction

Building a regression model for **multimodal data** (combining image and text features) poses unique challenges, especially with **high-dimensional embeddings** (700–2000 features) and a **small dataset**. The goal is to predict two independent continuous target variables with maximal accuracy. This report surveys state-of-the-art approaches for such tasks and provides practical recommendations. We cover modern **neural network architectures** and **libraries** for multimodal regression, strategies for handling **continuous target distributions** (including normalization and fragment-wise aggregation), techniques for **regularization and data augmentation** to combat overfitting, and preprocessing methods for managing **high-dimensional inputs**. The emphasis is on recent (2023–2024) research-backed practices, organized into clear sections for easy reference.

Modern Architectures for Multimodal Regression

Achieving high accuracy on multimodal regression requires choosing the right model architecture to fuse image and text embeddings effectively. Current state-of-the-art architectures range from simple feed-forward models to advanced transformers:

- **Early Fusion MLP:** The simplest approach is to **concatenate image and text embeddings** into one feature vector, then feed it into a Multi-Layer Perceptron (MLP) regressor. This early-fusion strategy treats all features uniformly. It's easy to implement and often a strong baseline, but it may not capture complex cross-modal interactions.
- **Late Fusion (Two-Tower Networks):** Another approach is to use **separate sub-networks for each modality** (e.g. a small dense network for image embedding and another for text), and then merge their outputs (by concatenation or element-wise operations) in a later layer. This allows each modality to be processed and scaled in its own space before fusion. It can improve performance by preserving modality-specific patterns and then learning an interaction in the fusion layer.
- **Transformer-Based Fusion:** Recent research explores using attention mechanisms to fuse modalities. **Cross-modal transformers** can treat the image and text embeddings as a sequence of tokens processed with self-attention, or apply cross-attention where one modality attends to the other. These architectures (inspired by vision-language models for VQA and captioning) can learn fine-grained interactions between image and text features. For example, **Perceiver IO** and related transformer models are modality-agnostic and can ingest combined multimodal inputs via latent attention, scaling to high-dimensional data efficiently. Such models have achieved strong results on vision-language tasks by learning a joint representation space . However, transformers typically require more data; with limited data, one might need to leverage pretraining or heavy regularization to avoid overfitting.

- **Pretrained Multimodal Models:** Where data is scarce, leveraging **pretrained vision-language models** is a cutting-edge strategy. Models like **OpenAI's CLIP, Meta's FLAVA, or BLIP-2** come pre-trained on huge image-text datasets and learn a shared embedding space for vision and language. For instance, CLIP learns image and text encoders jointly by predicting which caption matches an image, yielding state-of-the-art image-text representations . These representations can be directly used as features or fine-tuned for regression. Fine-tuning a large multimodal model (with a regression head) can greatly boost performance by transferring learned knowledge, as shown in recent studies. The downside is increased complexity and the risk of overfitting if fine-tuning on very few samples – techniques like freezing most layers or using **adapter layers** (parameter-efficient fine-tuning) can help in those cases.

Comparison of Multimodal Model Architectures:

Architecture	**Description**	**Pros**	**Cons**
Early Fusion MLP	Concatenate all 700–2000 dims into an MLP.	Simple; few parameters.	May ignore modality-specific patterns; limited interaction modeling.
Two-Tower Network (Late Fusion)	Separate networks for image vs text, then combine features.	Preserves modality-specific feature processing; flexible fusion (concat or weighted).	More parameters than early fusion; still mostly linear fusion.
Transformer-Based Fusion	Use self-attention or cross-attention to fuse embeddings.	Captures complex interactions; order-invariant processing (if designed for sets).	Data-hungry; higher complexity; needs regularization on small data.
Pretrained Multimodal (e.g. CLIP, FLAVA)	Fine-tune or use pre-trained vision-language encoders with a regression head.	Leverages learned representations; often high accuracy even with limited new data.	High computational cost; risk of overfitting if not properly fine-tuned; requires careful adaptation.

Modern multimodal toolkits facilitate building these models. **PyTorch** and **TensorFlow/Keras** support multi-input models (e.g., using the Functional API in Keras or custom `forward` in PyTorch to handle image and text inputs). Higher-level libraries like **Hugging Face Transformers** provide classes (such as `VisionTextDualEncoderModel` or CLIPModel) that bundle a vision encoder and text encoder, which can be jointly fine-tuned for a downstream task. Meanwhile, **AutoML frameworks** have started supporting multimodal data: for example, **AutoGluon Multimodal** offers a `MultiModalPredictor` that can take image and text features and internally ensemble several models to predict regression or classification targets with minimal coding. Similarly, **Uber Ludwig** (an open-source declarative Deep Learning framework) allows users to train a multimodal model by simply specifying which inputs are images and text, abstracting the architecture details (using CNNs, RNNs/transformers under the hood). These libraries streamline experimentation with different architectures and often implement best practices (like early fusion vs late fusion) by default.

Multimodal Frameworks and Tools:

Library/Toolkit	Capabilities for Multimodal Regression	Notable Features
PyTorch (w/ TorchVision & HuggingFace)	Full flexibility to construct custom multimodal models (e.g., combine a CNN and Transformer).	Pretrained models (e.g. CLIP) available; need to write training loop or use PyTorch Lightning.
TensorFlow/Keras	Easy multi-input model definition via Functional API.	High-level layers for concatenation, etc.; supports mixed data types natively.
Hugging Face Transformers	Pre-built vision+text models (CLIP, ViLT, etc.) that can be fine-tuned for regression.	Large model zoo; `Trainer` API simplifies fine-tuning; uses pre-trained multimodal embeddings.
AutoGluon Multimodal	AutoML for tabular, text, image data combined; can train regression models on multimodal inputs.	Ensembling of models under the hood; hyperparameter tuning; minimal code (just pass data).
Uber Ludwig	Declarative config-based training on multimodal data (images,	No coding needed for model architecture; uses proven architectures

Library/Toolkit	Capabilities for Multimodal Regression	Notable Features
	text, etc.) including regression tasks.	internally; provides evaluation and visualizations.

Using these frameworks, practitioners can experiment with architecture choices quickly. For instance, one could start with an early-fusion MLP in Keras as a baseline, then try HuggingFace’s CLIP model fine-tuned with a regression head, and compare results. Given no constraints on inference time, one might even ensemble multiple approaches (e.g., an MLP and a transformer model) to squeeze out extra accuracy, albeit at the cost of complexity.

Balancing Continuous Targets and Fragment Aggregation

When predicting continuous targets, especially two independent variables, careful consideration is needed for their distribution and how the data is labeled:

Normalization and Scaling of Targets

It is best practice to **normalize continuous target variables** before training. This can involve standardizing to zero-mean, unit-variance or scaling the targets to a fixed range (e.g. [0,1]). Normalization helps the neural network learn more effectively by ensuring the losses are on a comparable scale for each target. If the two target variables have very different ranges or units, scale them individually – this prevents one target’s variance from dominating the loss. In the case of a **multi-output regression**, a common approach is to use a weighted loss or adjust learning rates if one target is inherently more variable than the other. For example, using an **uncertainty-weighted loss** (as in multi-task learning research) can automatically balance the contribution of each output to the total loss based on their observed variance. At minimum, monitor both targets’ errors separately to ensure the model is fitting both well.

Imbalanced Continuous Distributions

If the distribution of the continuous targets is skewed or imbalanced (certain value ranges are underrepresented), the model may bias towards the more frequent ranges. Traditional class balancing techniques don't directly apply since there are no discrete classes . Recent research on **deep imbalanced regression** highlights that continuous targets lack clear boundaries between classes, making imbalance trickier to handle . One solution is to perform **target value transformations** – for example, apply a log or power transform if the target is heavy-tailed, so that differences in the high range are compressed. Another approach is **continuous oversampling or reweighting**: one can bucket the continuous target into bins and treat them like pseudo-classes for balancing. Data points from sparse regions of the target space can be upsampled (replicated more often in training) or given higher loss weights. There are also specialized techniques like **SMOGN (SMOTE for Regression)** which generate synthetic samples in regression tasks by interpolating feature-target pairs, focusing on underrepresented target ranges. Advanced methods propose **distribution smoothing**, which means instead of each sample only training on its exact target value, the loss is computed against a "smoothed" target distribution that acknowledges nearby target values . This was shown to calibrate the learned feature distribution to better cover the entire target range . In practice, a simpler form is to use a **Huber loss (smooth L1)** or quantile loss which is less sensitive to outliers, effectively giving a more robust training on skewed data.

For continuous targets that are **independent**, ensure that the model's output layer has two neurons (one per target) with no constraint tying them together (e.g., do not use a softmax, since these are regression outputs). If the targets are on different scales, the network's outputs can be on different scales too; just remember to invert the normalization when interpreting predictions.

Fragment-Wise Aggregation and Multi-Instance Learning

The problem description mentions that targets are "averaged over fragments," implying each training example may be composed of multiple fragments (e.g., an image split into patches or a long text split into segments) and the target is an average of some per-fragment values. In such cases, a straightforward approach is to **aggregate fragment embeddings** before prediction – for example, average all fragment embeddings to form a single representation for the example, and use that as input to the regression model. However, averaging can wash out important variation between fragments. A more sophisticated approach is to use a **multiple instance learning (MIL)** paradigm: treat each fragment as an instance and the whole example as a “bag” of instances. The model can then learn to predict the bag’s label (the averaged target) from all instance embeddings. Techniques like **attention pooling** can learn weights for each fragment’s contribution to the final prediction (so the model can up-weight more relevant fragments) instead of a simple mean. This is analogous to MIL in which a network learns to aggregate information from a set of instances to predict a bag-level output. For example, an **attention-based MIL network** can output a learned weighted average of fragment features, focusing on those that correlate with higher target values. If the number of fragments per example varies, models like the Set Transformer (which is designed to handle set inputs of varying size) or permutation-invariant pooling operations can be used. This fragment-wise modeling can capture the distribution of fragment values, not just their mean, which sometimes leads to better accuracy than using pre-averaged features.

In practice, if implementing fragment-wise training is complex, one compromise is **data augmentation by fragment shuffling or grouping**: for each training example composed of fragments, create multiple training samples by randomly selecting subsets of fragments and using the average of those subsets (which should theoretically still equal the overall average in expectation). This increases the training sample count slightly and might improve generalization. Nonetheless, the safest bet for simplicity is to compute the aggregate embedding (mean or concatenation of fragment features) and ensure the target is normalized appropriately for that level of aggregation.

Regularization and Data Augmentation Strategies

With limited data and high-dimensional inputs, **overfitting** is a primary concern. The following regularization strategies and augmentation methods can significantly improve generalization:

Regularization Techniques

- **Dropout:** Incorporate dropout layers in the network, especially in the dense layers of the regressor. Dropout randomly zeros out a fraction of neurons during each training step, which forces the network to learn redundant representations and prevents over-reliance on any one feature. This “noise injection” is proven to reduce overfitting by preventing co-adaptation of neurons. For instance, a dropout rate of 0.2–0.5 is common. Dropout can even be applied to the input layer (sometimes called “feature dropout”) – randomly dropping some input features (or entire embedding components) during training can improve robustness given the high dimensionality.
- **Weight Regularization:** Use L2 weight decay on network parameters to discourage large weights (which often indicate overfitting to noise). This adds a penalty to the loss for large weights and tends to make the model coefficients smaller and more general. In practice, one might add an L2 penalty (e.g., 1e-4 to 1e-6) on dense layers. L1 regularization could be used to encourage sparsity (which might effectively select a subset of the 700–2000 features if many are irrelevant), though L1 is less common than L2 for neural nets.
- **Early Stopping and Cross-Validation:** With few samples, setting aside a validation set and monitoring its loss is crucial. Use early stopping to halt training when validation error starts rising, thus capturing the model at the point of best generalization. Additionally, **k-fold cross-validation** can be employed to make full use of the small dataset: train multiple models each leaving out a different fold as validation, then either choose the best or average their predictions (this also gives an ensemble effect that often improves accuracy).
- **Smaller Network or Shared Layers:** While the drive might be to use a complex model to capture multimodal interactions, a simpler model can generalize better on small data. Consider reducing the number of layers or hidden units if overfitting remains an issue. Another trick for multi-output (two targets) is to let the two outputs **share the majority of network layers**, only splitting into two separate output layers at the end. This effectively acts like a form of regularization (multi-task learning) since the model must find representations that serve both targets. Even if the targets are independent in theory, in practice this can be beneficial when data is limited – it’s forcing the network to learn common patterns if any.

- **Batch Normalization / Layer Normalization:** Including normalization layers can stabilize training and have a mild regularization effect. Batch Norm, in particular, adds noise when batch sizes are small (due to sampling variance in mean/std computation), which can act similarly to dropout. It can be helpful after the first dense layer that follows the concatenation of features, to mitigate differences in scale among features.

Data Augmentation

Even with fixed embeddings as inputs, data augmentation can be applied at the raw data level or in feature space to effectively enlarge the dataset:

- **Image Augmentation:** If you have access to the original images (before embedding), apply standard augmentation techniques: random crops, flips, rotations, scaling, color jitter, etc. These create new images that are semantically similar, and you can recompute their embeddings via the image encoder. This increases the variety of image embeddings the model sees. Recent research and practice show that strong image augmentation can improve generalization significantly on small datasets. Ensure that the augmented image's target label remains the same (since these transformations shouldn't change the underlying output on average).
- **Text Augmentation:** Similarly, augment text data (if raw text is available) using techniques like synonym replacement, random insertion/deletion of words, or back-translation (translating to another language and back to English). These **Easy Data Augmentation (EDA)** methods have been shown to boost performance on text tasks with limited data by injecting variability. The resulting augmented text can be passed through the text encoder to get new embeddings. For example, replacing some words with synonyms or paraphrasing a description should not fundamentally change the regression target, but will yield a slightly different text embedding, helping the model not to overtly memorize exact text embeddings.

- **Mixup and Feature Space Augmentation:** When working directly with embeddings, one powerful augmentation is **Mixup**. Mixup involves taking two training examples and linearly interpolating both their features and their targets. For instance, take embedding A (with target y_A) and embedding B (with target y_B), and create a new training sample: `embedding_mix = λ * A + (1-λ) * B` and `target_mix = λ * y_A + (1-λ) * y_B`, where λ is a random number in [0,1]. This creates synthetic points in the feature-target space that smooth the model's mapping function . Mixup is known to act as a regularizer and can especially help in small data regimes by informing the model about intermediate target values between known data points. Since our targets are averages over fragments, mixing two samples' embeddings and targets is logically plausible (the mix is an average of two averages). Besides mixup, even simpler: adding slight Gaussian noise to embeddings during training (feature noise injection) can help the model generalize local perturbations.
- **Fragment-wise Augmentation:** If each data point has multiple fragments, one can augment by **shuffling or subsetting fragments** as mentioned earlier. Also, if sequence matters (say a text split into parts), one might randomize the order of fragments (if order is irrelevant) to generate new combinations. This is a bit task-dependent, but the key idea is to **get creative in generating additional training examples** that are consistent with the known data distribution.

It's worth noting that **data augmentation for regression** should be done with care – ensure that any synthetic variation you introduce does not alter the expected output. For images and text, human judgement or domain knowledge is needed to know what transformations preserve the target. In feature space, augmentations like mixup assume linearity between features and targets, which, while not strictly true, tend to smooth the function the network learns and can improve performance.

By applying a combination of these regularization and augmentation techniques, even a complex model can be trained on a small dataset without severe overfitting. For example, you might train a transformer-based fusion model with heavy dropout and L2 regularization, while also augmenting images and text – the net effect is a model that is both powerful (due to architecture) and robust (due to regularization).

Preprocessing and Network Design for High-Dimensional Inputs

The input feature vector (700–2000 dimensions, from concatenated image+text embeddings) is relatively large, especially given a small dataset. Proper preprocessing and network design can ensure efficient learning:

- **Feature Standardization:** Just as we normalize targets, we should **normalize input features**. If the image and text embeddings come from different encoders, they may have different scales or distributions. It's common to apply z-score normalization to each feature (subtract mean, divide by std dev) using the statistics from the training set. At the very least, ensure the embeddings are centered around 0 – many transformer-based embeddings (like CLIP or BERT) are not naturally zero-centered. Standardizing can help gradient descent converge faster and avoid one modality overpowering the other (for instance, if image embeddings have larger magnitude than text embeddings).
- **Dimensionality Reduction:** With limited data, using all 1000+ features might introduce a lot of noise and spurious correlations. Consider reducing the dimensionality of the combined embeddings. One way is to apply **Principal Component Analysis (PCA)** on the feature set to compress it (e.g., reduce 1000 dims to 200) while retaining most variance. This can act as a denoising step. Alternatively, incorporate a learnable reduction in the model: for example, use a dense layer with fewer units than input features as the first layer – this forces the network to learn a lower-dimensional representation of the inputs. This technique is similar to an autoencoder's bottleneck; it prevents the model from simply memorizing each feature and encourages generalization. Recent studies often use a **projection layer** on pre-trained embeddings (e.g., projecting CLIP embeddings to a 256-dim space) when training downstream models, finding that it can improve performance by focusing on the most salient aspects of the embedding .
- **Modality-Specific Preprocessing:** If we know the makeup of the 700–2000 dimensions (say 512 from an image model, 768 from a text model, concatenated), it can help to preprocess each part separately. For instance, you could L2-normalize the image embedding and text embedding separately (especially if using cosine similarity-trained embeddings like CLIP, which often are used with L2 norm). You might also drop or down-weight dimensions known to be less informative. Some practitioners even perform **feature selection** using simple filters (like taking the top N features by correlation with the target) as a preprocessing step, then feed only those features to the network. This is more common in classical ML, but it could be tried here to see if a subset of embedding dimensions carries most signal.

- **Efficient Network Design:** To handle high-dimensional input without exploding the number of parameters, keep the first hidden layer relatively small. For example, if input is 1000-dim, a first layer of 128 units (with nonlinear activation) will have 128k weights, which is reasonable. In contrast, if one naively used a wide network (say 1000 -> 1000 units), that's 1e6 weights just in the first layer, likely overfitting instantly on a small dataset. Another design is the **two-tower network** described earlier: e.g., project the 512-dim image embedding to 128 and the 512-dim text embedding to 128 in two separate branches (each branch has 512x128 weights), then concatenate the 128+128=256 features and maybe have one more layer to 64 and then to outputs. This drastically cuts down parameters versus a single-layer on full 1024 features. It's also easier for the network to adjust one modality's influence – if text is less important, that branch can learn to output smaller activations, etc.
- **Handling Two Outputs:** For the final output layer, you can use a fully connected layer with 2 neurons (no activation, if predicting real values directly). In many cases, a single output layer with 2 units is fine for two independent regression targets. If the targets are on very different scales, you could train two separate output layers (each preceded by its own small dense if needed) – but usually, sharing the majority of the network and just having two linear outputs works well and is simpler. Ensure the loss function is appropriate – typically **Mean Squared Error (MSE)** or **Mean Absolute Error (MAE)** for regression. Sometimes a custom loss per target (like if one target is more critical, you weight its MSE higher) can be used.
- **Outlier Handling:** If some data points have extreme target values (outliers), consider clipping the targets or using a robust loss (as mentioned, Huber loss). Preprocessing the target by winsorizing (clamping extreme values) or even removing a couple of outliers might significantly help the model focus on the relevant range. With small data, one outlier can skew training a lot.

In summary, treat the high-dimensional embedding input with similar care as any high-dimensional dataset: normalize features, reduce dimensionality if possible, and use network architectures that don't blow up the parameter count upfront. These steps ensure the model training is efficient and mitigates the curse-of-dimensionality effect where too many features relative to samples can lead to overfitting.

Conclusion and Practical Recommendations

Designing a multimodal regression model under these conditions requires balancing complexity with generalization. **First**, leverage **pretrained embeddings and models** whenever possible – they provide a head start in feature quality. Build on them with a modestly complex model (start with an MLP or two-tower network; only move to transformers if necessary). **Second**, invest in **preprocessing**: normalize everything (inputs and outputs), consider dimensionality reduction, and carefully think through how fragment-level data should be aggregated. **Third**, address any imbalance in target distributions – if one range of values is rare, use data or loss-based balancing to avoid bias . **Fourth**, apply **regularization liberally** (dropout, weight decay, early stopping) and use **data augmentation** to enrich your training set (augment raw data if you can; use mixup or noise on embeddings if not). These practices are supported by recent research and competitions: for example, imbalanced regression techniques from 2021 show improved performance with distribution smoothing , and transformer-based multimodal models from 2022–2023 demonstrate the power of aligned image-text representations even for new tasks . Finally, don’t hesitate to **experiment** – with no constraints on inference time, you can ensemble different approaches (averaging predictions from an MLP and a fine-tuned transformer, for instance) to further boost accuracy. By following these guidelines, one can craft a proof-of-concept multimodal regressor that makes the most of limited data and yields reliable predictions on both continuous targets.

Sources