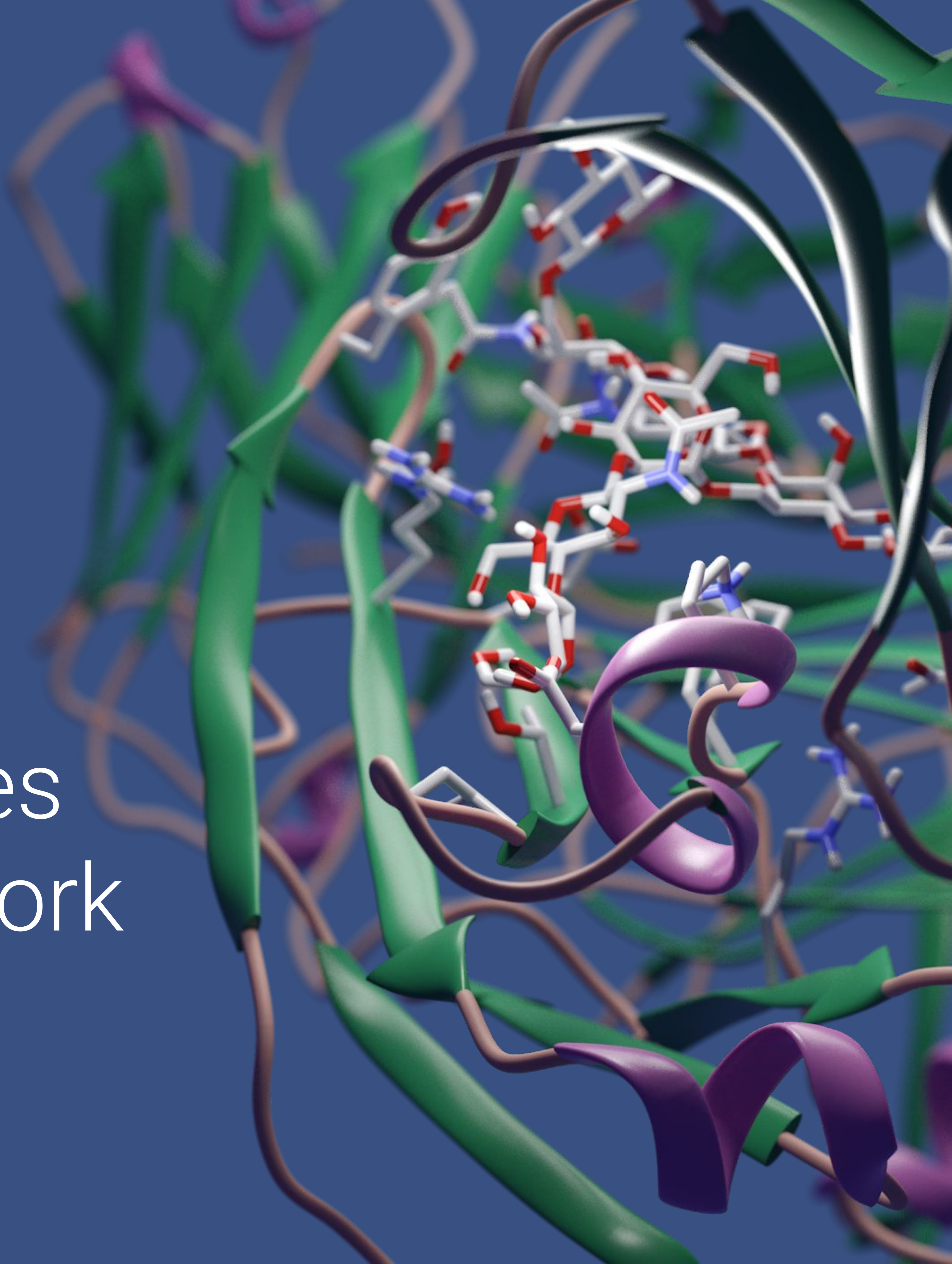
TOTIENT

Predicting molecular properties with Graph Transformer Network

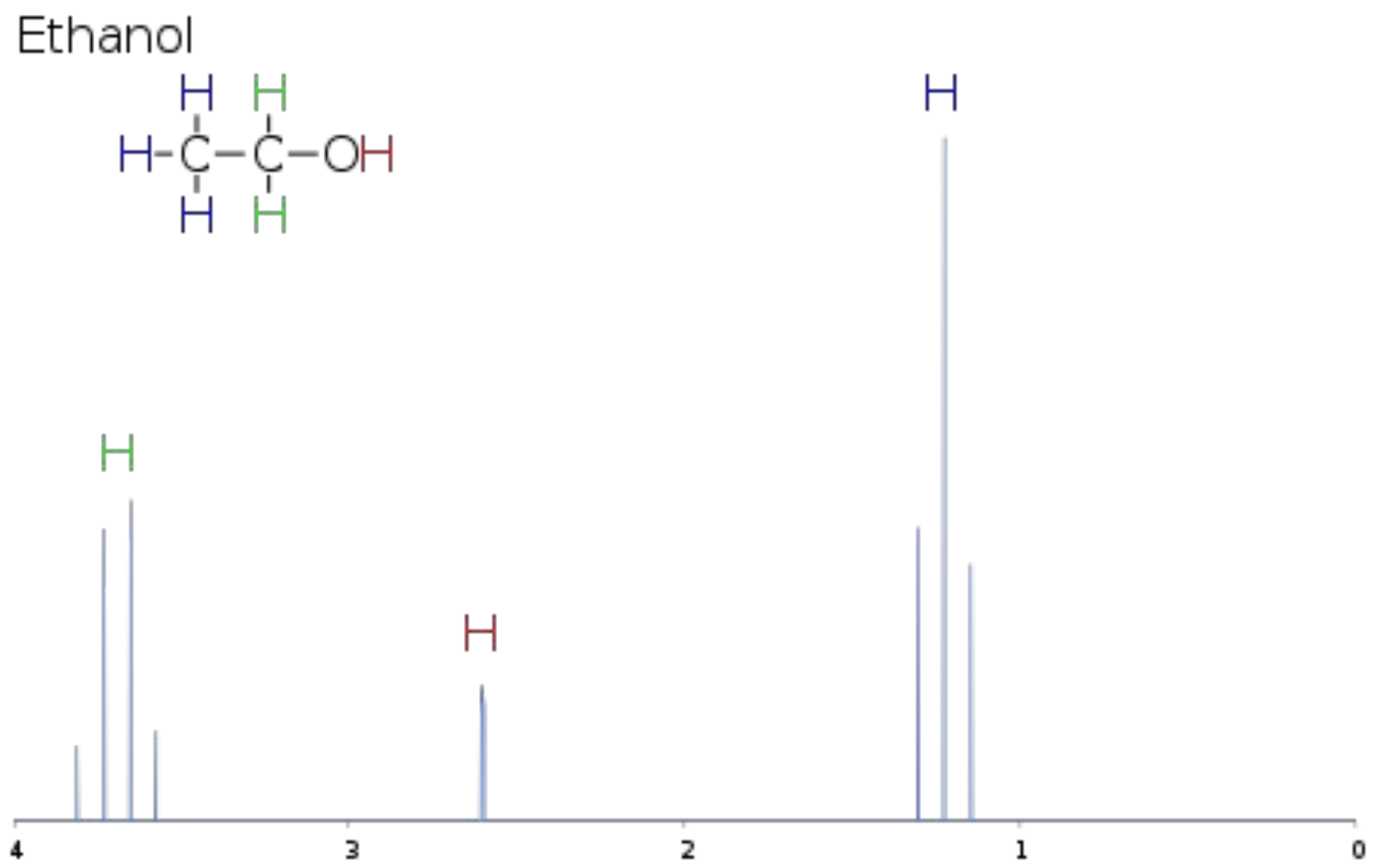


About the competition that started it all

- Title and tagline:
  - Predicting Molecular Properties
  - Can you measure the magnetic interactions between a pair of atoms?
- Organized by **CHAMPS** (**CH**emistry **A**nd **M**athematics in **P**hase **S**pace)
  https://champsproject.com/
- Task:
  - Given atom coordinates in 3d space, predict "Scalar Coupling Constant" (a specific type of magnetic interaction) between designated atom pairs
- Notes:
  - It is possible to calculate SCC using QM methods
  - QM methods are very expensive (days or weeks per molecule) and have limited applicability in day-to-day workflows
  - Top 5 teams get to write a paper with CHAMPS group

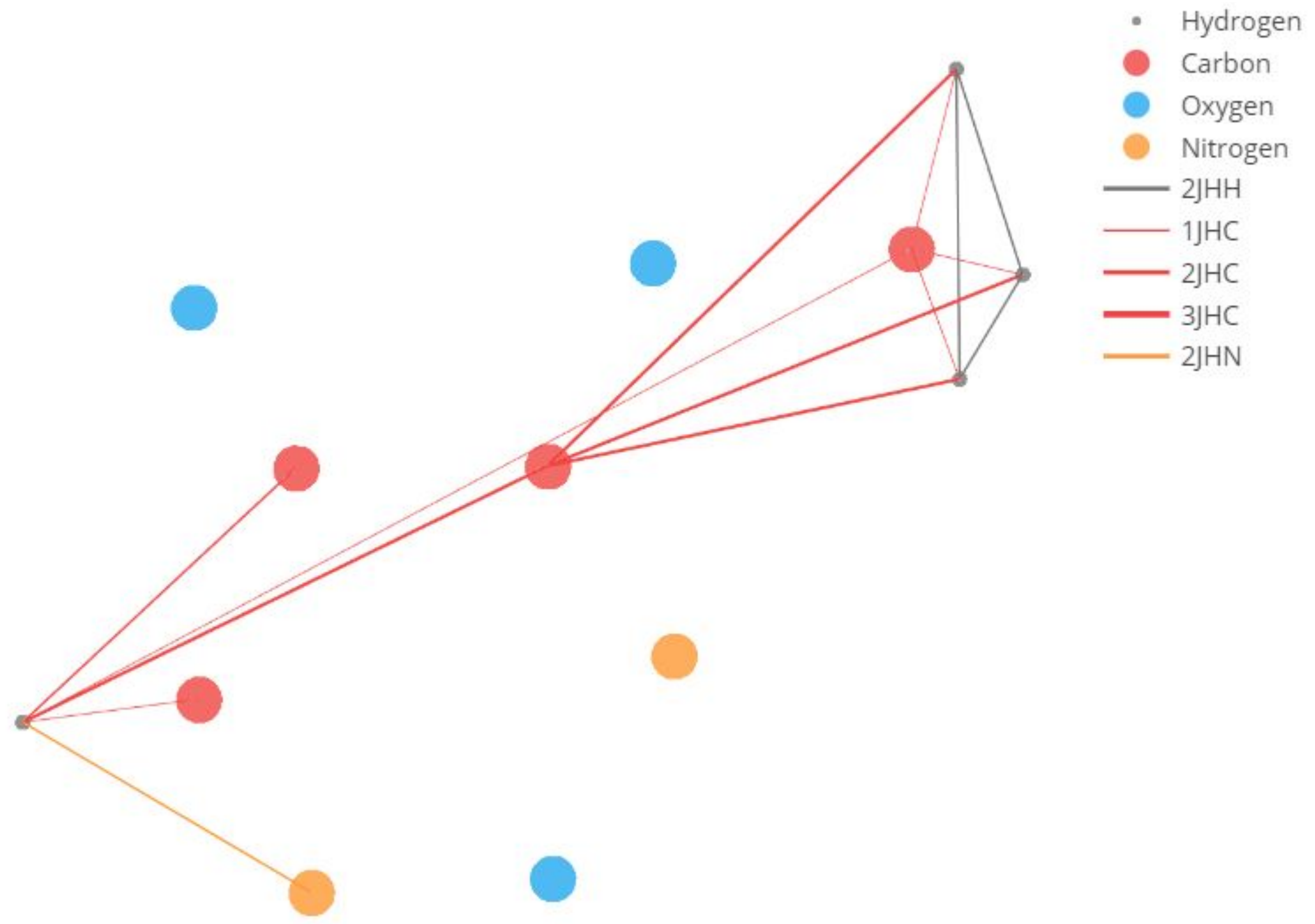
About NMR and Scalar Coupling Constant

- Nuclear magnetic resonance (NMR) spectroscopy is used to identify structure of chemical compounds
- SCC between atoms determines relative positions of peaks on a spectral diagram
- SCC varies depending on the types of atoms, theirs distances, angle between the bonds and other structural properties
- Task was to predict SCC between pairs of H atoms, H-C and H-N pairs that are either directly bonded or 2 and 3 bonds apart



Input Data

- Dataset consists of **~85.000 molecules**.
- Up to 30 **H, C, O, N, F atoms** per molecule (up to 9 "heavy" - non H atoms)
- Every molecule is represented as a list of **atoms described by atom type** and **x, y, z coordinates** - no bonds or other info
- **Labels are provided as** a list of molecule ID, pair of atom IDs, **coupling type** (2JHH, 3JHH, 1JHC, 2JHC, 3JHC, 1JHN, 2JHN, 3JHN) **and the scalar coupling constant**



Scoring Function

- Log of the Mean Absolute Error, calculated for each scalar coupling type, averaged across types, so that a 1% increase in MAE for one type provides the same improvement in score as a 1% increase for another type

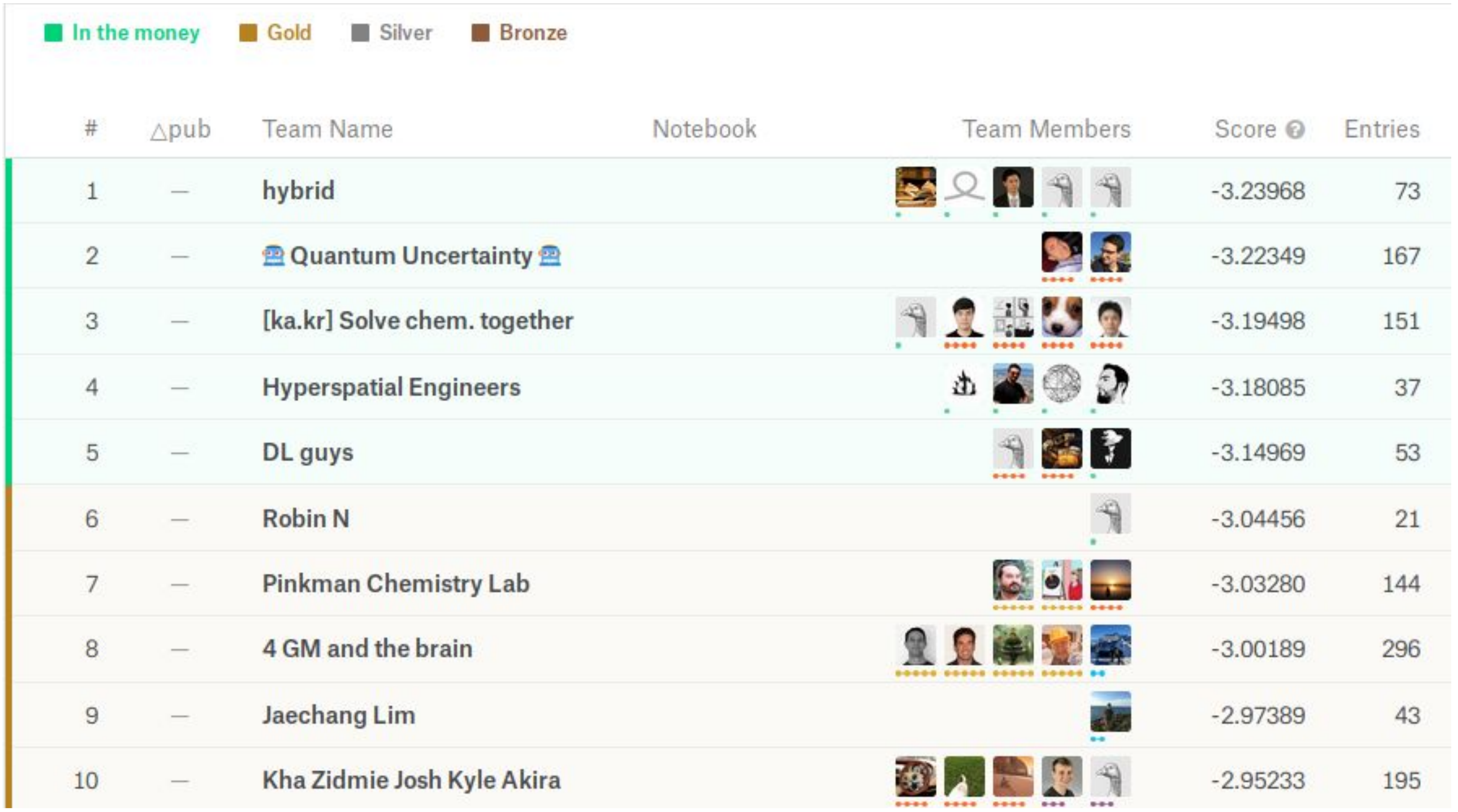
$$score = \frac{1}{T} \sum_{t=1}^{T} \log\left(\frac{1}{n_t} \sum_{i=1}^{n_t} |y_i - \hat{y}_i| \right)$$

Where:

- T is the number of scalar coupling types
- n_t is the number of observations of type t
- y_i is the actual scalar coupling constant for the observation
- $\hat{y}_i$ is the predicted scalar coupling constant for the observation

Results

- 2,749 teams
- #4 placement
- All top solutions are DNNs
- Many competition masters and grand masters
- Team #1
  - very strong domain expertise in both ML and QM
  - Lots of computational resources
  - Bosch Corporate Research and Bosch Center for AI (BCAI), CMU



🟩 In the money 🟨 Gold ▨ Silver 🟫 Bronze

#	△pub	Team Name	Notebook	Team Members	Score	Entries
1	—	hybrid			-3.23968	73
2	—	Quantum Uncertainty			-3.22349	167
3	—	[ka.kr] Solve chem. together			-3.19498	151
4	—	Hyperspatial Engineers			-3.18085	37
5	—	DL guys			-3.14969	53
6	—	Robin N			-3.04456	21
7	—	Pinkman Chemistry Lab			-3.03280	144
8	—	4 GM and the brain			-3.00189	296
9	—	Jaechang Lim			-2.97389	43
10	—	Kha Zidmie Josh Kyle Akira			-2.95233	195

Deep Graph Library Overview

- https://www.dgl.ai/
- PyTorch for working with graph NNs
- Supports directed graphs only
- Both nodes and edges have associated vector/tensor properties
- Relies on message passing paradigm

$$m_{src,dst} = f(h_{src}, h_{dst}, e_{src,dst})$$

$$h_u = \mathop{A}_{i=1}^{N_u} m_{v,u}$$

h_u – embedding of node u

$e_{u,v}$ – embedding of edge connecting nodes u and v

$m_{src,dst}$ – message from source node to destination node

N_u – neighborhood of node u

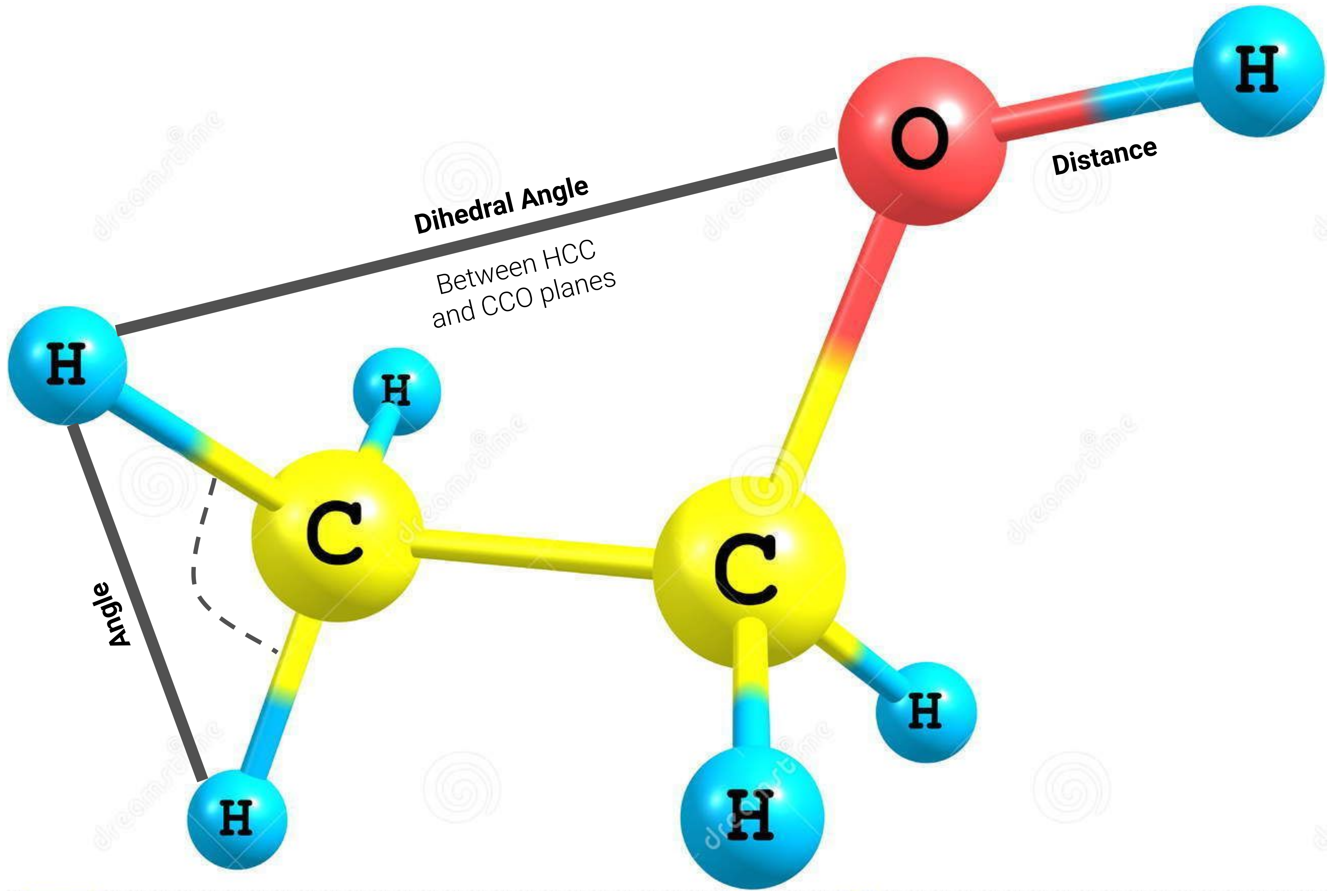
A – arbitrary aggregation function

Our data representation

- To apply a graph neural networks we need to construct a graph from the points:
  - Atoms become nodes
  - For edges we:
    - Inferred bonds using OpenBabel, cleaned-up using custom scripts
    - Inserted artificial 2 and 3 jump edges
    - Inserted artificial self edges
- Node data:
  - N-dimensional embedding vector for atom type (H, C, O, N, F)
  - Electronegativity, first ionization energy, electron affinity, mulliken charge)
- Edge data:
  - N-dim edge type embedding (single, double, triple, self, 2jump, 3jump)
  - Distance (all edges), angle (2jump only), dihedral angle (3jump only)
    - All standardized to zero mean unit variance

Our data representation

Enables rotational and translational invariance



Model

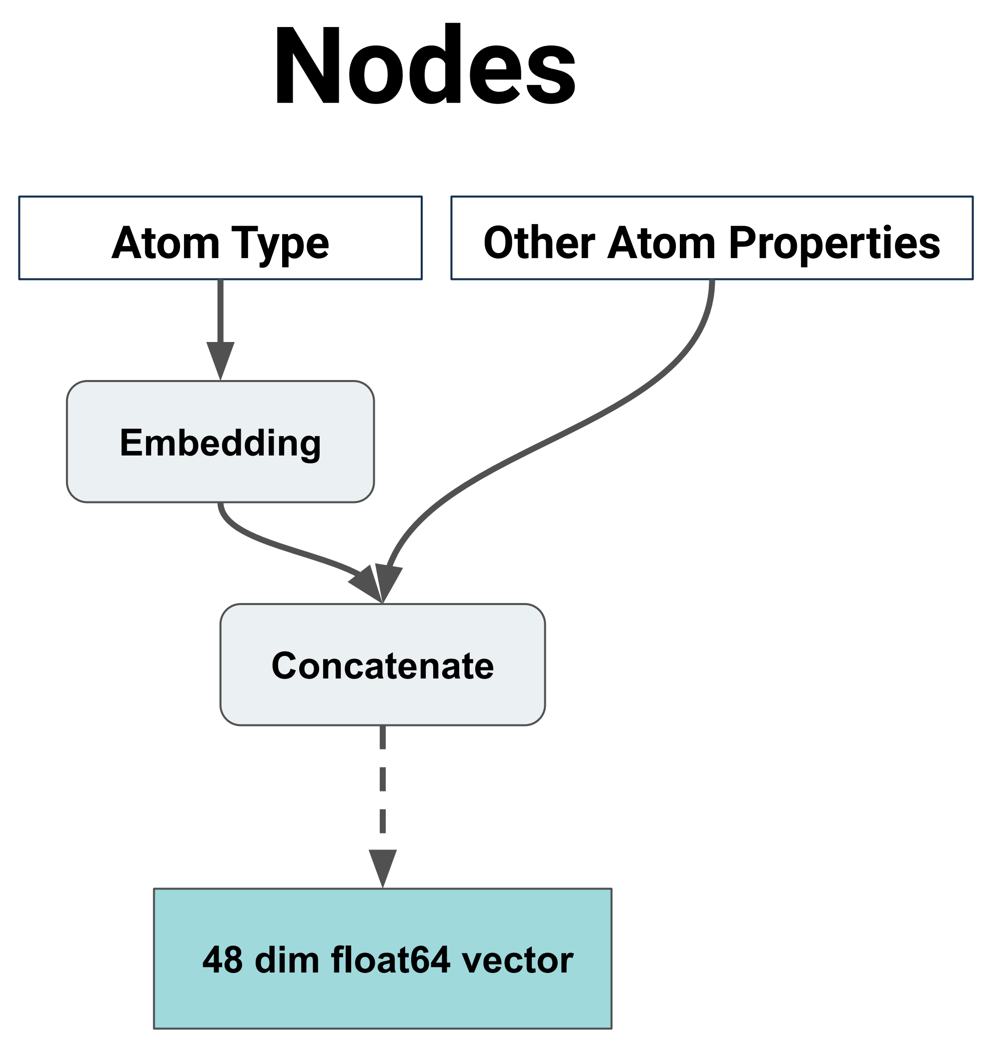
- Inspired by Transformers described in Attention Is All You Need and GAT (Graph Attention Networks, Petar Velickovic et al.)
- One regression output per coupling type
- Minimized mean MAE loss instead of mean log(MAE)
- 57.8 million parameters to optimize

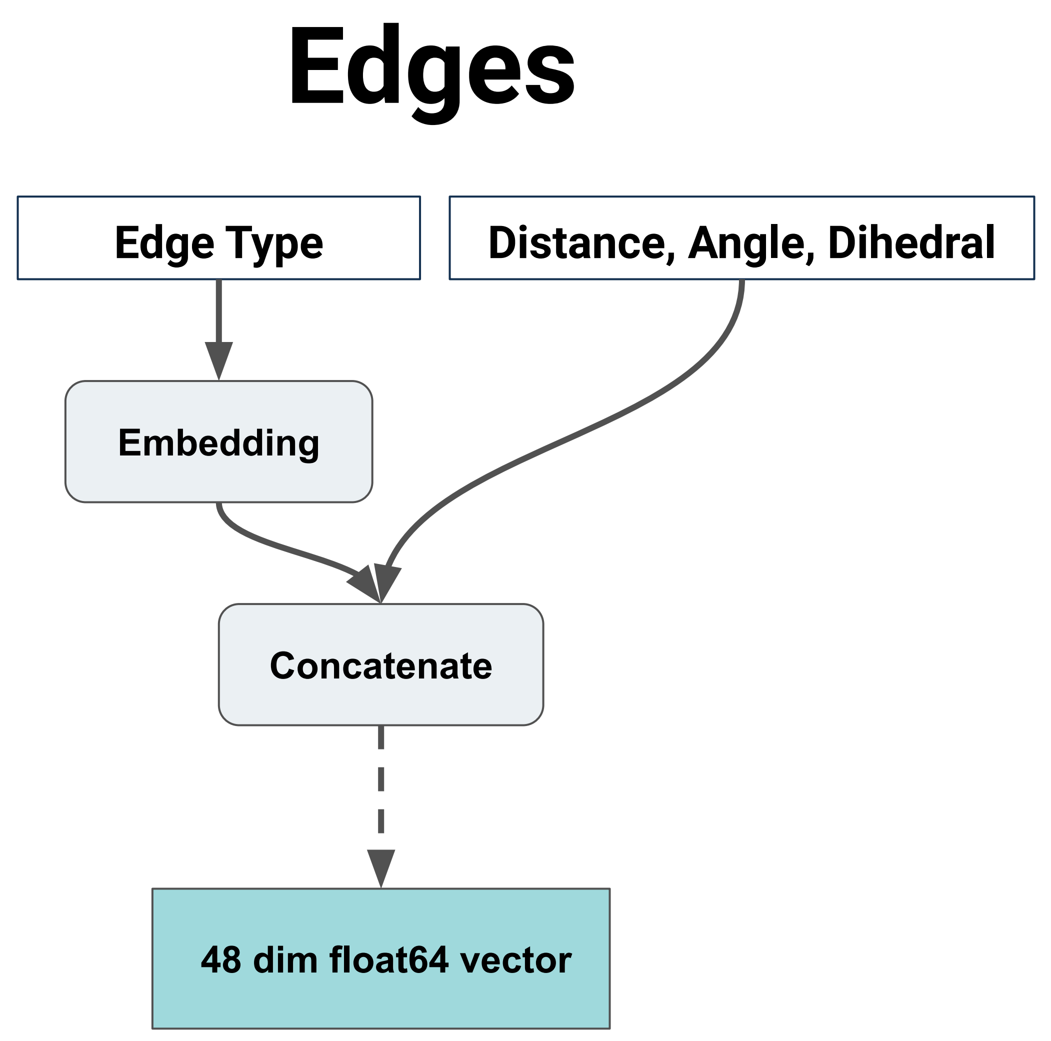
$$score = \frac{1}{T} \sum_{t=1}^{T} \cancel{\log} \left(\frac{1}{n_t} \sum_{i=1}^{n_t} |y_i - \hat{y}_i| \right)$$

Embeddings

- Used to embed categorical variables into an N-dim vector space
- Method is very simple:
  - Randomly initialize an CxN matrix where C is the number of categories, while N is the chosen embedding vector dimensionality
  - Use these vector representations in your model
  - Allow the optimizer to optimize the matrix
- The optimization process will tend to cluster the vectors of similar categories, specifically similar in terms of properties relevant to training task

Node and Edge Embeddings





Graph Attention Idea

- Graph Attention:
  - All node embeddings are first transformed with a parameterized function of your choice creating a “message”
  - New embedding for node X is an aggregation (sum) over all messages from neighbours (including X), weighed by an “attention coefficient” which is calculated by:
    - Applying a parameterized function of your choice to a concatenation of the pair of messages
    - Normalizing the results of this function to sum to 1 over all neighbours
- Multi-head Attention:
  - Same as above, but with multiple (N) attention coefficients and N different node embedding transformation functions
  - The resulting N embeddings are then either reduced using mean or sum reduction, or concatenated
- Problem with this approach
  - Doesn’t include edge properties in the update

Our Multi-Head Attention

- Incorporates edge embeddings into the process
- We've used 24 attention heads and 48d embeddings resulting in a 1152d vector representation of each node and edge

$$m_{src,dst} = lin(h_{src}) * lin(e_{src,dst})$$

$$\alpha_{src,dst} = leaky\_relu(lin(h_{dst}||e_{src,dst}||h_{src}), 0.2)$$

$$\alpha'_{src,dst} = softmax(\alpha_{src,dst}) \text{ over neighbors}$$

$$h_{dst} = \sum_{u \in N_{dst}} \alpha'_{u,dst} * m_{u,dst}$$

$$e_{src,dst} = lin(h_{src}||e_{src,dst}||h_{dst})$$

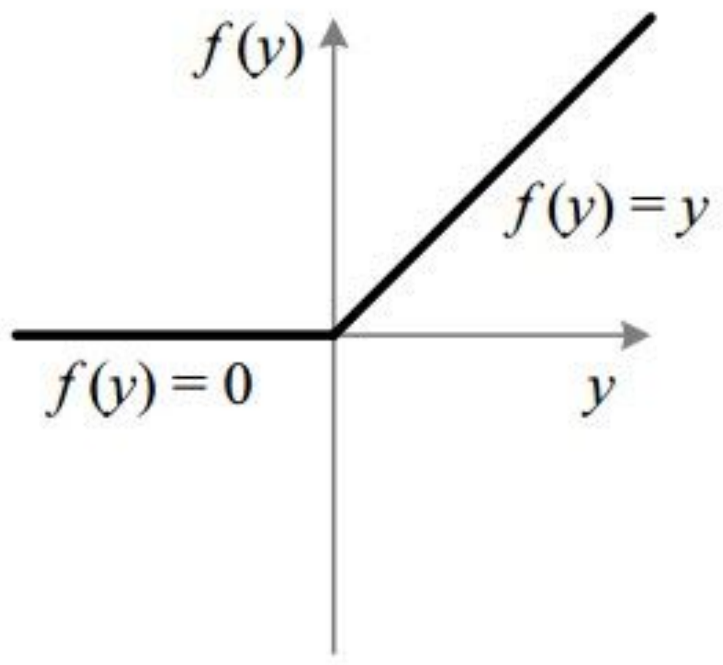
Softmax

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

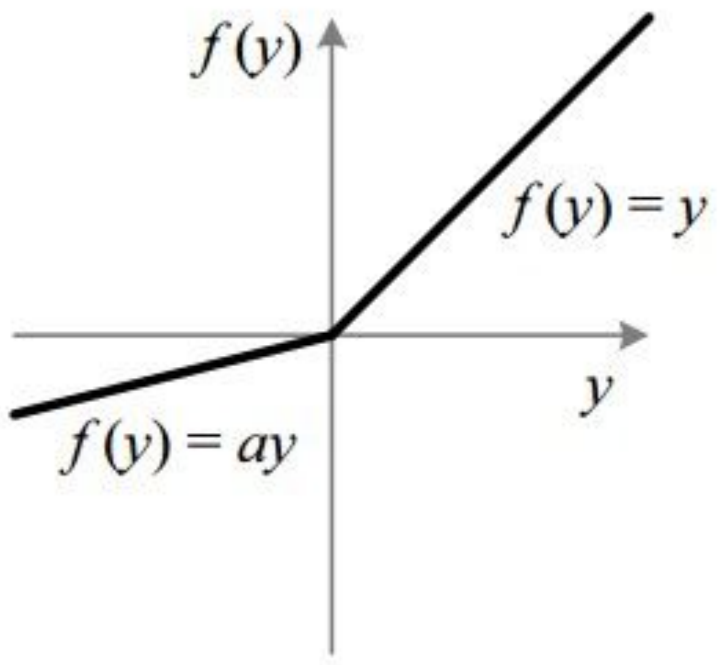
ReLU, LeakyReLU and pReLU

- Most commonly used non-linearity today is Rectified Linear Unit (ReLU)
- The derivative of ReLU is 0 for negative inputs, creating a ‘dead ReLU problem’
- LeakyReLU was created to mitigate this problem, by having a small negative derivative alpha
- pReLU is a parametric LeakyReLU where alpha is a trainable parameter

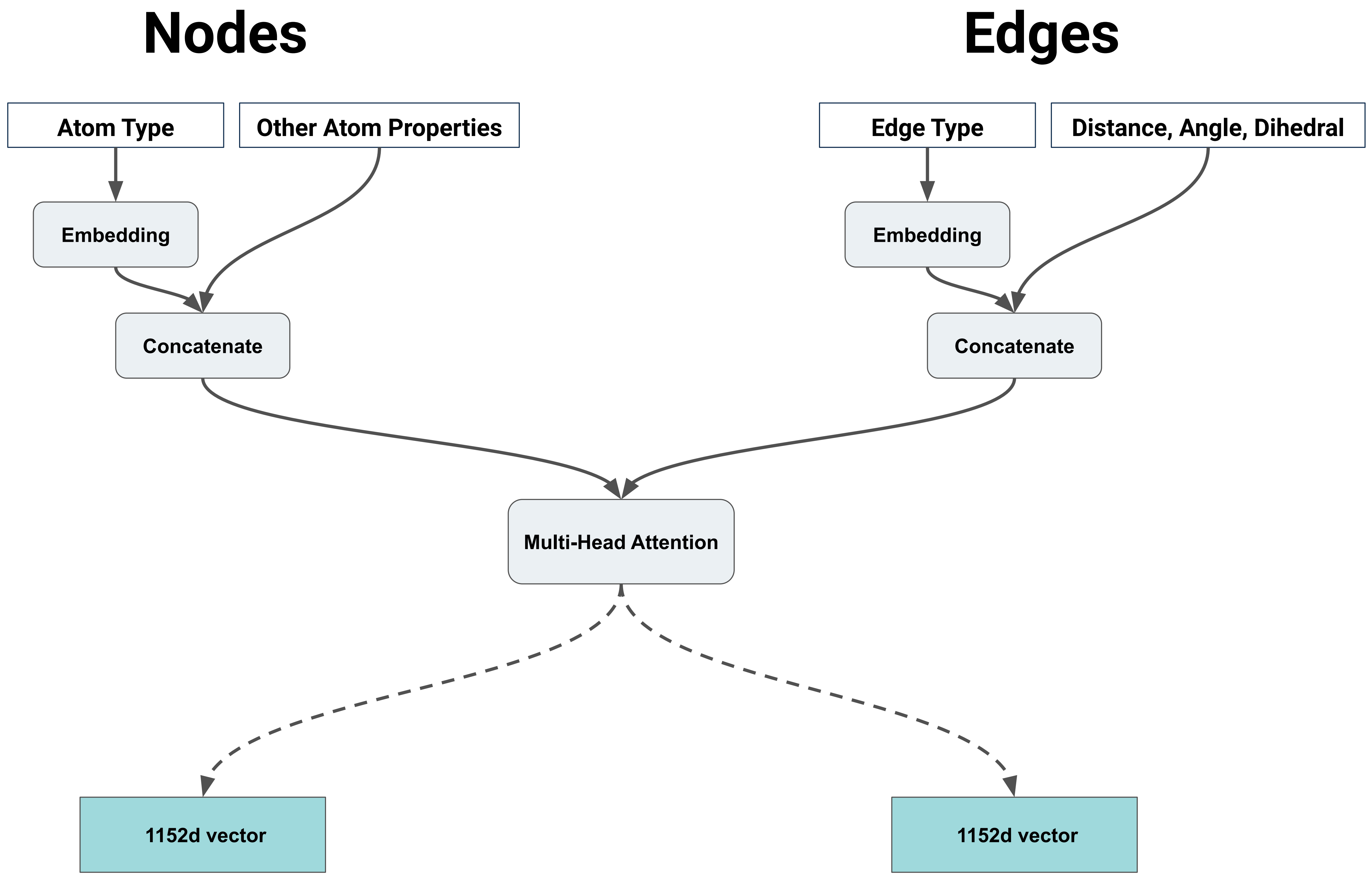
ReLU



LeakyReLU



Attention



Nodes
Edges
Atom Type
Other Atom Properties
Edge Type
Distance, Angle, Dihedral
Embedding
Embedding
Concatenate
Concatenate
Multi-Head Attention
1152d vector
1152d vector

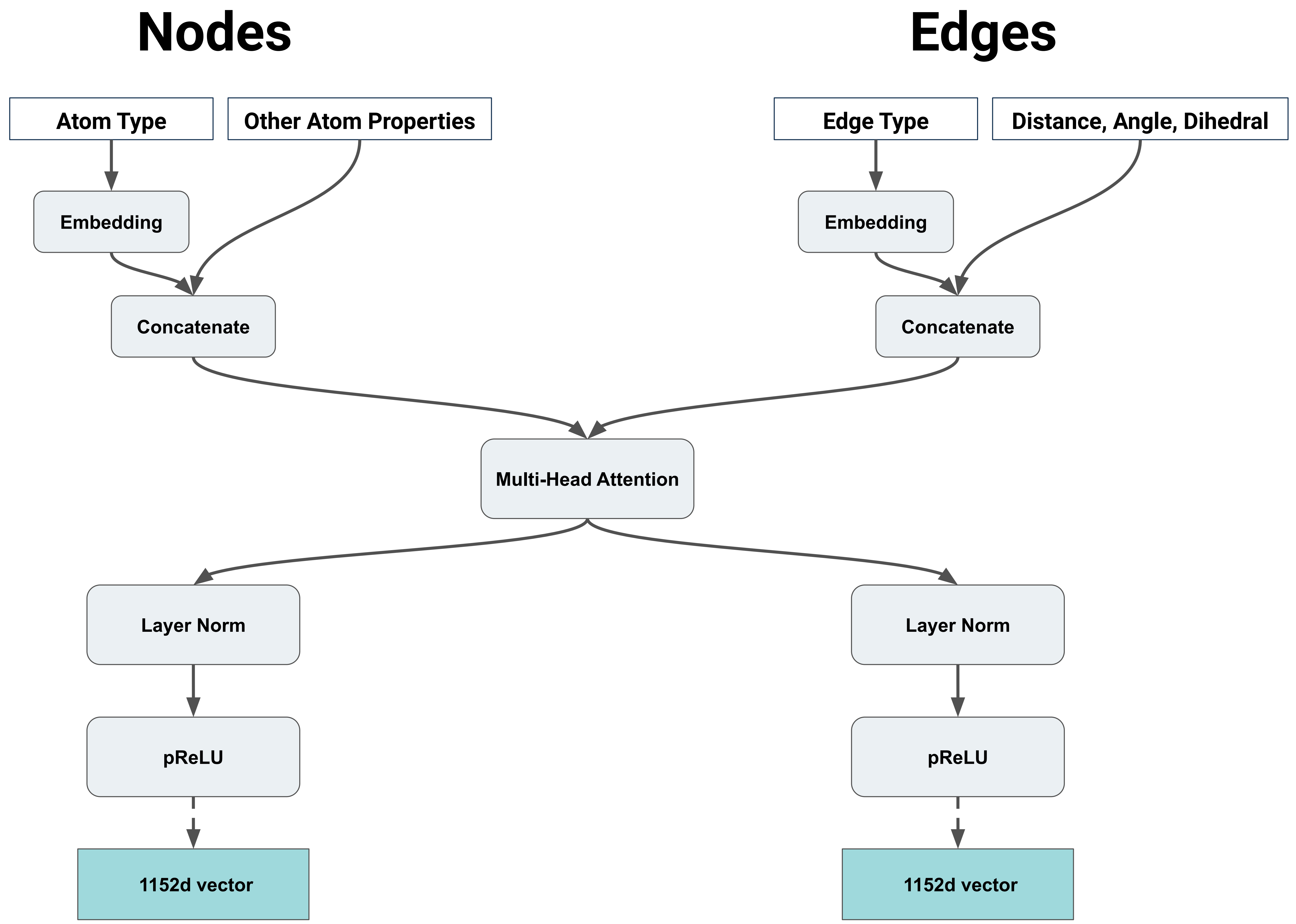
Layer Norm

- Layer Normalization; Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton
- In deep networks, neuron activations tend to die out or explode making it hard to train them
- Activation normalization between layers can help stabilize the network
- Layer Norm normalizes inputs to a layer to zero mean unit variance
- After normalization, Layer norm applies scaling and shifting using two trainable parameters

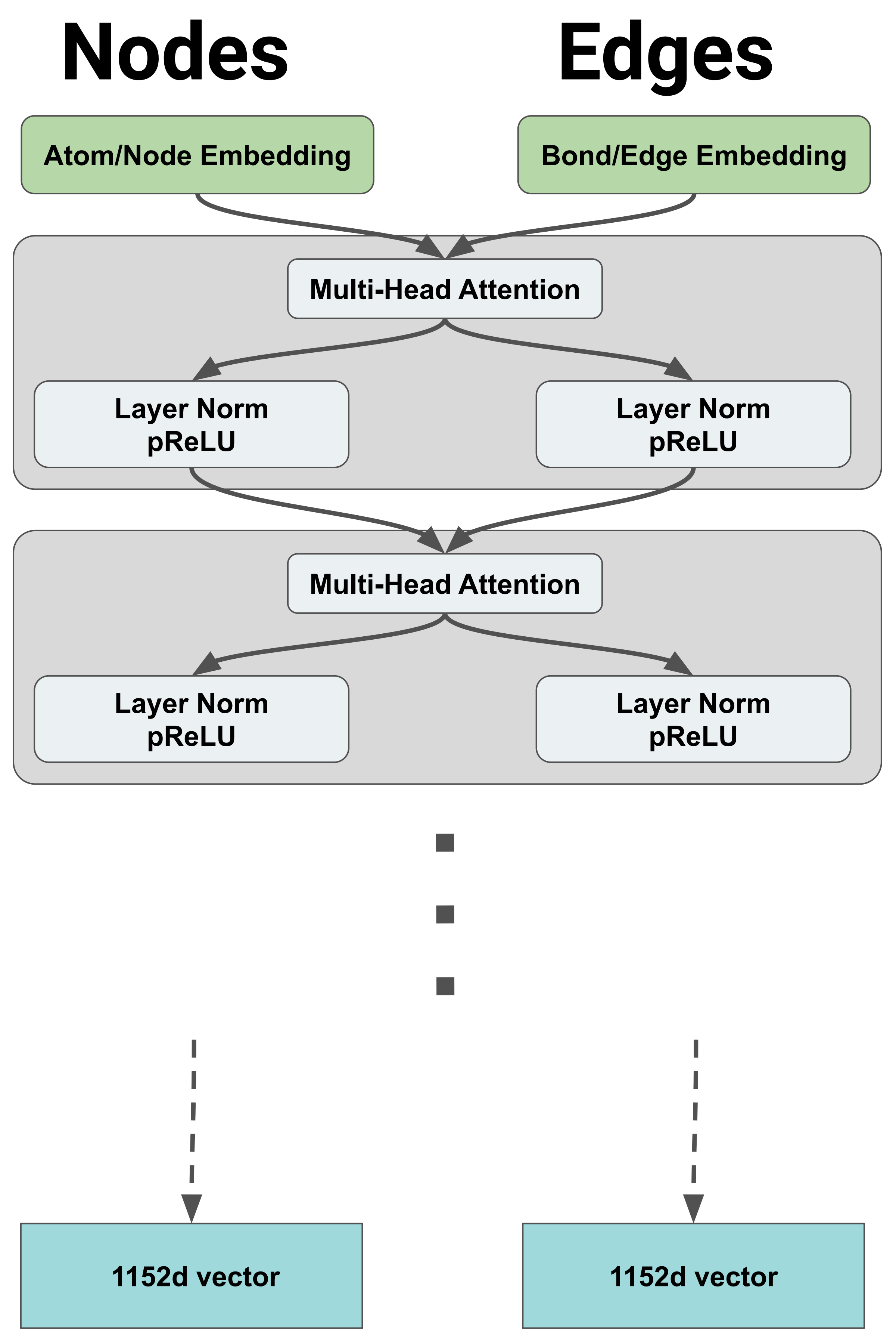
$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

***H* - Number of neurons in the layer**
***l* - Layer**
a - Activation from the previous layer (input the current layer)

Normalisation and non-linearity



Stacked attention blocks

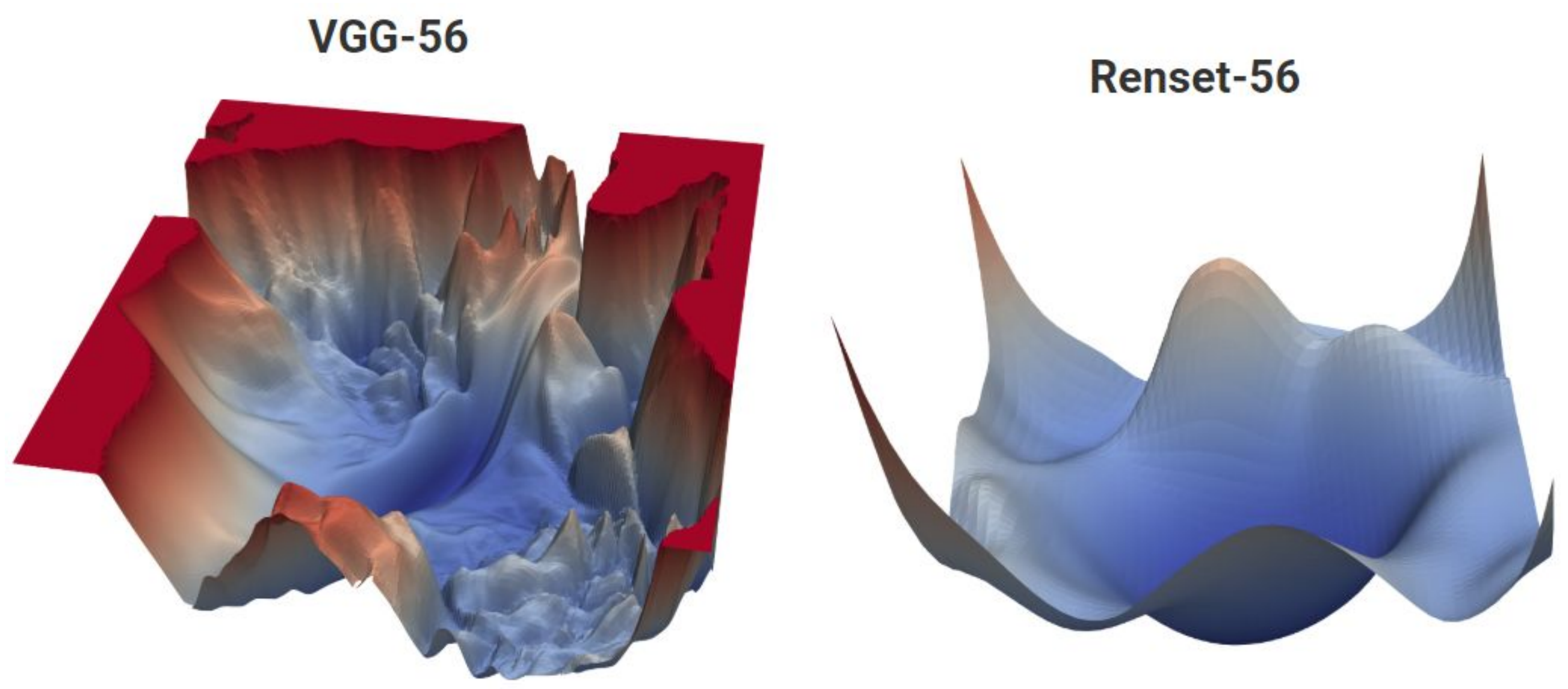


Residual connections

- Very deep NNs (VDNNs) suffer from diminishing gradients as we move back towards first layers of the network, causing significant training slowdown
- Residual (skip) connections sum the output of a layer with it's inputs

$$r(f(x)) = x + f(x)$$

- Skip connections create a shorter path for gradients to flow through, thus mitigating the diminishing gradients problem
- It was also observed that loss surfaces in VDNNs are very rough
- Same network architectures with skip connections have much smoother loss surfaces (Visualizing the Loss Landscape of Neural Nets)



Gated Residual connections

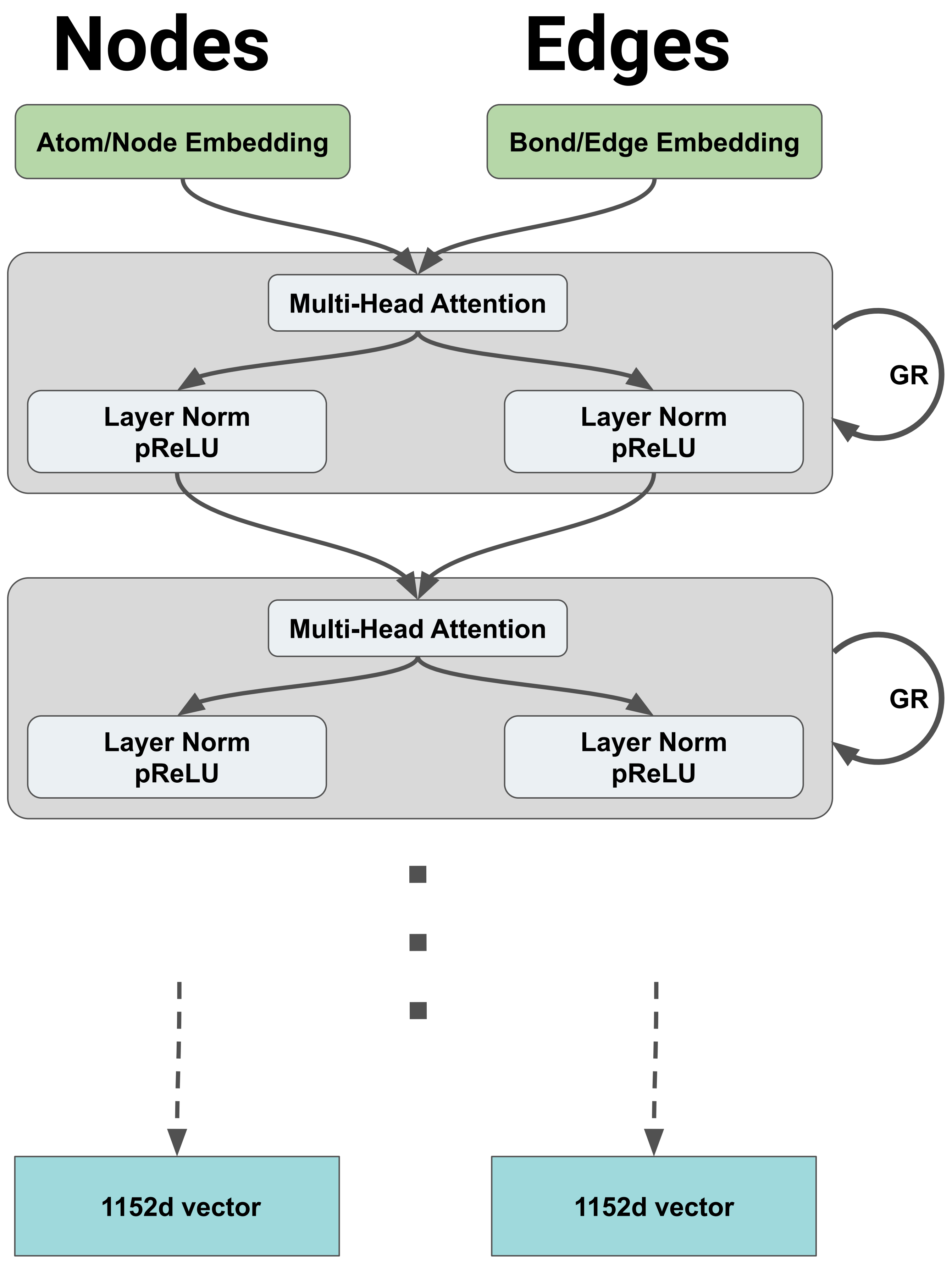
- However it was observed that Deep Residual Networks diminish in accuracy with as the number of layers increases
- To mitigate this issue Gated Residual connections were proposed, which are a linear interpolation between **x** and **f(x)** with trainable interpolation factor

$$z = \sigma(lin(x))$$

$$gr(f(x)) = z * x + (1 - z) * f(x)$$

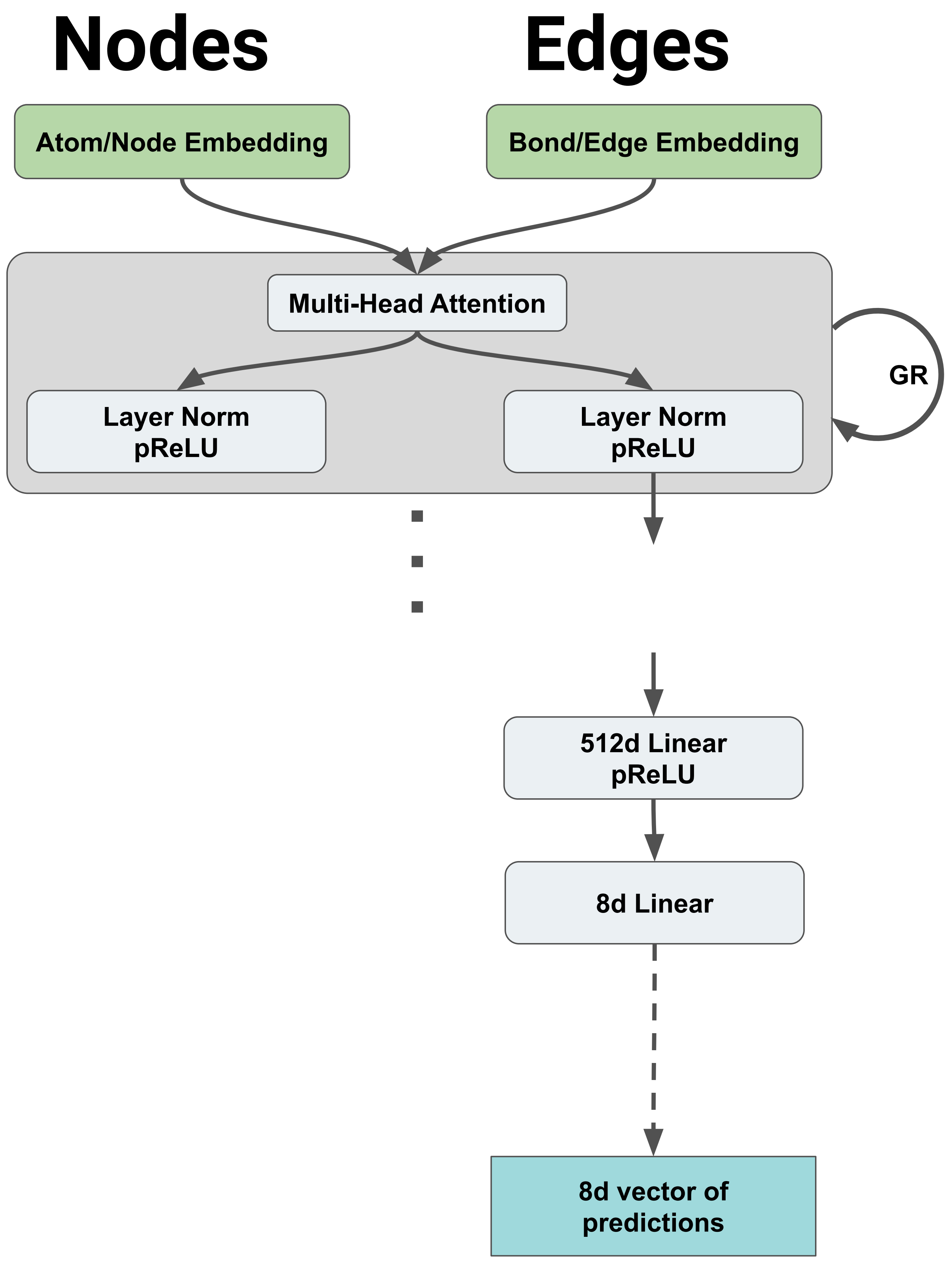
- These have been shown to perform much better and start diminishing in accuracy with a higher number of layers/blocks

Stacked attention blocks



Output layers

- Since we have artificial 2 and 3 jump edges we can read predictions directly off of edges
- 8-dim output, one output per SCC type
- Edges with no labels do not contribute to loss
- Our graphs are directional, and we have edges in both directions, so we get 2 predictions per atom pair
- 2 predictions are independently minimized in training, but are averaged to get a final test prediction acting as a "mini-ensemble"



LAMB optimizer and LAMBW

- Large Batch Optimization for Deep Learning: Training BERT in 76 minutes
- Uses ADAM as the base
    - ADAM is SGD with momentum, with adaptive learning rate per parameter
- Controls the size of updates by introducing trust ratio:

$$\beta = \frac{||\theta||}{||\Delta_\theta||}$$

$$\theta = \theta_{t-1} - \alpha * \beta * \Delta_{theta}$$

- LAMBW
    - LAMB implementation includes weight decay **(-wd*theta)** into the update vector directly, so WD is included into the trust ratio calculation and is attenuated by it
    - Our modification LAMBW (inspired by ADAMW), removes WD from the update and applies it post hoc

$$\theta = \theta_{t-1} - \alpha * (\beta * \Delta_\theta - \delta * \theta)$$

Stochastic Weight Averaging

- Averaging Weights Leads to Wider Optima and Better Generalization
- DNNs make a tight fit to the training set so small statistics shift in the test set could result in bad accuracy, but how and why?
- Wide and flat minima have been shown to generalize better
- Once SGD converges it “bounces” around the minimum, landing on the slope of a minimum
- This leaves some directions in parameter space with very steep loss increases, so any shift along those directions could cause large drops in accuracy
- SGD “bouncing” can be interpreted as sampling parameters from the surface of a sphere centered at the minimum with radius proportional to the learning rate
- If we keep the learning rate constant, and sample enough parameter vectors we can average them to get to the center of the sphere reaching the flatter area less prone to errors with small shifts

Stochastic Weight Averaging

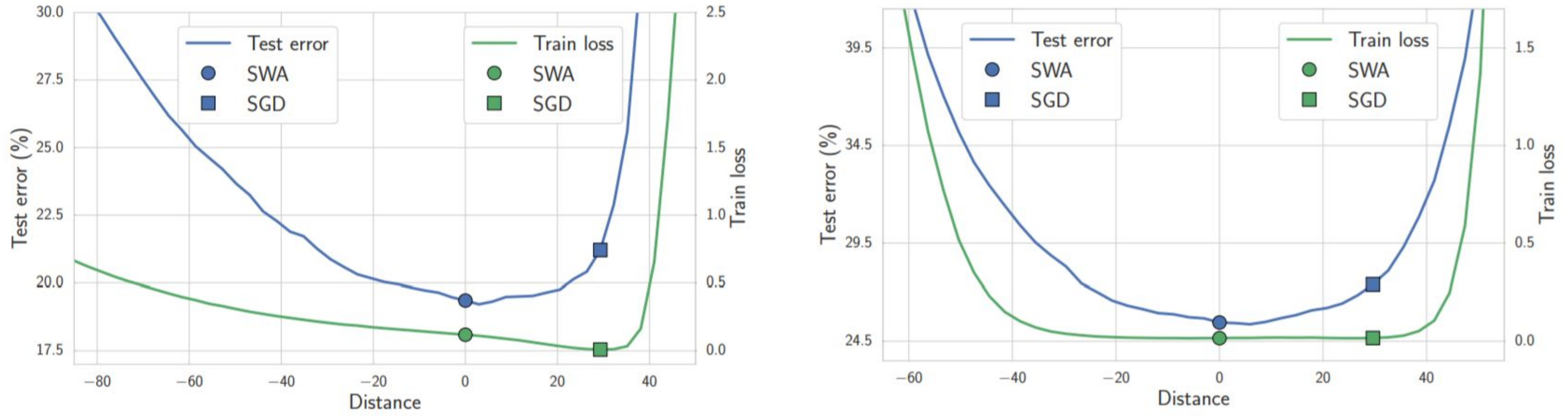


Figure 5: L_2-regularized cross-entropy train loss and test error as a function of a point on the line connecting SWA and SGD solutions on CIFAR-100. **Left**: Preactivation ResNet-164. **Right**: VGG-16.

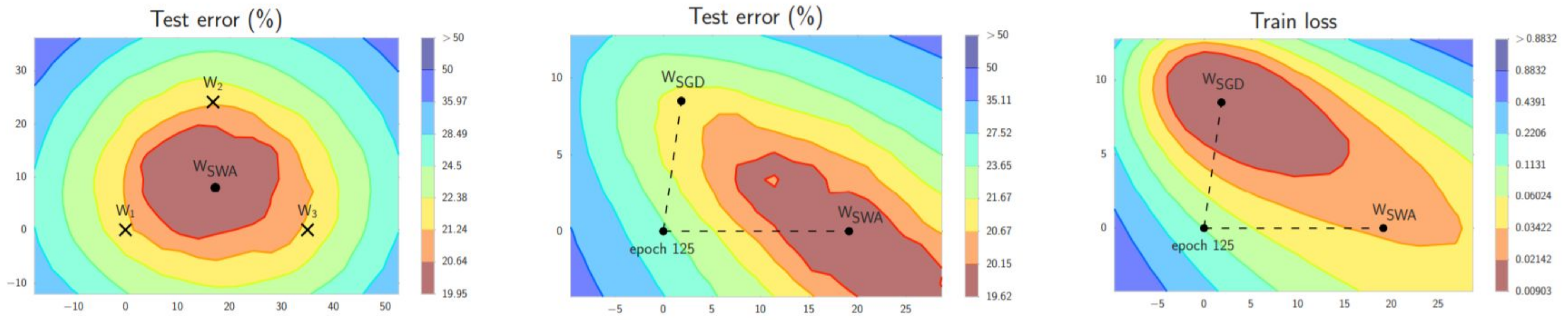


Figure 1: Illustrations of SWA and SGD with a Preactivation ResNet-164 on CIFAR-100[1]. **Left**: test error surface for three FGE samples and the corresponding SWA solution (averaging in weight space). **Middle** and **Right**: test error and train loss surfaces showing the weights proposed by SGD (at convergence) and SWA, starting from the same initialization of SGD after 125 training epochs.

Training Regime

- Overview:
  - Pre-trained one model for all SCC types, then fine tuned for each type
  - Training set split into 90%/10% train/eval split
  - Trained with 0 mean 1 variance target, as well as just shifted 0 mean
  - Entire procedure repeated twice with two random train/eval splits
- Training procedure:
  - Pre-train phase, optimizing mean MAE per type
    - 30 epoch LR cycle (1e-3 to 1e-2), 5e-2 weight decay (WD from now on)
    - Constant 1e-3 LR, 1e-2 WD until convergence (~70 epochs)
  - Fine-tune phase, optimizing MAE for one type
    - Constant LR 1e-3, 1e-2 WD until convergence (~100 epochs)

GPU Runtimes

Model	**Train time**	**Test time**	**Private/Public score**
Final	200 GPU hr	40 GPU min	-3.18085 / -3.18667
One model per type	~85 GPU hrs	20 GPU min	-3.13853 / -3.14362
One model per type, estimated Mulliken	~85 GPU hr	20 GPU min	-3.072 / -3.07331
Single model, estimated Mulliken	~24 GPU hr	4 GPU min	-2.96183 / -2.96443