# Learning ML/DL

Benjamin Li

2025

# 1  Introduction

This document serves as an aggregation of my progress towards gaining a more fundamental understanding of how many modern ML/DL models work. To understand it, an introductory background in linear algebra (matrix operations), calculus (derivatives), and probability is assumed. Corresponding code implementations will be made available in the future, and some helpful references are included for each section. This document is also a bit text-dense so I hope to add some of my diagrams/drawings later.

# 2  Multilayer Perceptron (Fully-Connected Neural Network)

## 2.1  Overview and Network Initialization

First, let's look at a specific example. Let's say we have 10 soccer players (and two pieces of info - **features** - about their record number of consecutive juggles and weight in kilograms) and we want to determine if they're going to be drafted to a soccer team. Mathematically, we are given a batched input $x$ of dimensions $(b, f)$, where $b$ is the number of entries and $f$ is the number of features. In our case, let's say it's $(10, 2)$.

$$x = \begin{bmatrix} 50 & 75 \\ 80 & 68 \\ 100 & 80 \\ \vdots & \vdots \\ 40 & 60 \end{bmatrix}, y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

We are trying to predict a vector $y$ of dimensions $(10, 1)$ where 1 indicates that a player is drafted and 0 indicates they are not drafted.

To do this, we construct a **fully-connected neural network**. This architecture consists of multiple **layers**, and within each layer multiple **neurons**. Thus, before constructing the network we initialize an array $n = [2, 3, 4, 5, 1]$ in which each index represents the number of neurons for the corresponding layer. Notably, the index 0 represents the INPUT dimension, and index 4 is the OUTPUT dimension (must be 1 since we're trying to predict one outcome). Finally, randomly initialize a list of weights and biases (more details later). At a high level, training neural networks involves two steps: **feedforward** and **backpropagation**, and these are the next two sections.

## 2.2   Feedforward Process

Each layer $i$ calculates the following:

$$A_i = \sigma(W_i * A_{i-1} + B_i)$$

where $A_i$ is the "activated" output of the layer, $W_i$ is a weight matrix, $B_i$ is the bias matrix, and $\sigma$ is some activation function (e.g. sigmoid, relu) that introduces non-linearity. The intuition is that the weight matrices will automatically learn how to extract meaningful features from the data. However, we'll also need the **unactivated** output later for backpropagation: $Z_i = W_i * A_{i-1} + B_i$. First, let's figure out how to initialize the weights. Notably, because the $W_i$ left-multiplies the $A_{i-1}$ matrix, the columns of $W_i$ MUST be the same as the rows of $A_{i-1}$ based on the rules of matrix multiplication. We can understand the dimensions of the weights by looking at what happens when you pass through the **first** layer of the network. We treat the original input X as the precursor activated value since there doesn't exist at this point.

$$A_1 = \sigma(W_1 * X + B_1)$$

We know that X has dimensions $(10, 2)$. But, remember our list of neuron numbers: $n = [2, 3, 4, 5, 1]$ - we somehow need to get our $(10, 2) \rightarrow (10, 3)$. Specifically, this can be accomplished by doing $(3, 2) * (2, 10) \rightarrow (3, 10)$. We must transpose, or swap the rows and columns of, our input matrix from $(10, 2) \rightarrow (2, 10)$. This also motivates our selection of weights as having dimensions $(n_i, n_{i-1})$. Similarly, $B$ is the bias matrix of dimensions $(n_{[i]}, b)$. In our example, $W_0, W_1, W_2$, and $W_3$ are initialized to: $(3, 2), (4, 3), (5, 4), (1, 5)$, respectively.

For the biases, we initialize $B_0$, $B_1$, $B_2$, $B_3$ to $(3,1),(4,1),(5,1),(1,1)$, respectively.

Notably, we have to **broadcast** B from (3, 1) to (3, 10) - i.e. adding the same bias term to all samples in B - to retain the same shape for the matrix. Therefore, $Z_1 = W_1 * X^T + B_1 \rightarrow (3,2) * (2,10) + (3,10) \rightarrow (3,10)$. Afterwards, simply calculate $A_1 = \sigma(Z_1)$. For future iterations, use the previous activation value as input to calculate the value of Z. Additionally, as mentioned earlier, it's a good idea to store all activated and unactivated values as they will be required during backpropagation. This means, $Z_2 = W_2 * A_1 + B_2 \rightarrow (4,3) * (3,10) + (4,10) \rightarrow (4,10)$. Then, apply the activation function to find $A_2 = \sigma(Z_2)$. Repeat this process for all layers.

## 2.3   Backpropagation

Now that we have some output $\hat{y}$ from our network, how are we supposed to tweak the weights and biases in such a way that allows us to accurately predict the ground truth vector $y$? Mathematically, we try to minimize some loss/cost function C with respect to weights and bias values. There are many choices available for C that are good for different kinds of problems. One such loss function is called binary cross-entropy loss - it looks like this:

$$C = -y * \log \hat{y} + (1 - y) * \log (1 - \hat{y})$$

Our goal is to calculate $\frac{dC}{dW_i}$ and $\frac{dC}{dB_i}$. This tells us how much cost is affected by the weights from that layer - once we know this information, we can move in the OPPOSITE direction of the gradient by some factor (learning rate, lr) to minimize the cost. In other words, we want to update the weights and bias this way: $W_i = W_i - lr * \frac{dC}{dW_i}$ and $B_i = B_i - lr * \frac{dC}{dB_i}$ (yes, this is programming notation). The first derivative to calculate is from the final weight matrix: $\frac{dC}{dW_3}$, found via the **chain rule**!

$$\frac{dC}{dW_3} = \frac{dC}{dA_3} * \frac{dA_3}{dZ_3} * \frac{dZ_3}{dW_3}$$

To get $\frac{dC}{dA_3}$, take the derivative of $C$ with respect to input. This turns out to be $-(\frac{Y}{\hat{y}} - \frac{(1-Y)}{\hat{y}})$, where $\hat{y}$ is equivalent to $A_3$. It is dimensions $(3, 10)$ - the same as Z. Assuming we know what $\frac{dC}{dA_3}$ is, $\frac{dA_3}{dZ_3}$ is just the derivative of the activation function, and $\frac{dZ_3}{dW_3}$ is just going to be the previous activation value at that index (because $Z_i = A_{i-1} * W_i + B_i$).

$$\frac{dC}{dB_i} = \frac{dC}{dZ} * \frac{dZ}{dB_i}$$

3

Recall that

$$Z_i = W * A_{i-1} + B_i$$

so taking the partial derivative with respect to $B_i$ yields $\frac{dZ}{dB_i} = 1$. Therefore, $\frac{dC}{dB_i}$ simplifies to $\delta$! Now, how do we figure out the derivatives with respect to the other weights?? They aren't directly related to the cost function, after all.

We use the chain rule AGAIN! First, find the derivative of the cost vs. post-activation value for the next layer (layer 2 in our case).

$$\frac{dC}{dW_2} = \frac{dC}{dA_3} * \frac{dA_3}{dZ_3} * \frac{dZ_3}{dA_2} * \frac{dA_2}{dZ_2} * \frac{dZ_2}{dW_2}$$

For the bias:

$$\frac{dC}{dB_2} = \frac{dC}{dA_3} * \frac{dA_3}{dZ_3} * \frac{dZ_3}{dA_2} * \frac{dA_2}{dZ_2} * \frac{dZ_2}{dB_2}$$

If we wanted to keep going, we would do the following. For the derivative of cost with respect to weight, omit the last operation ($\frac{dZ_2}{dW_2}$) and multiply $\frac{dZ_2}{dA_1} * \frac{dA_1}{dZ_1} * \frac{dZ_1}{dW_1}$. Similar idea with bias: omit $\frac{dZ_2}{dB_2}$ and replace with $\frac{dZ_2}{dA_1} * \frac{dA_1}{dZ_1} * \frac{dZ_1}{dB_1}$.

For each layer, update the weights/biases using the new derivative values: $W_i = W_i - lr * \frac{dC}{dW_i}$ and $B_i = B_i - lr * \frac{dC}{dB_i}$. Ok, that was a lot... thankfully a lot of libraries such as PyTorch have autodifferentiation features which can automatically update our parameters with a lot less hassle. **But it's important to know that not all loss functions are differentiable!**

## 2.4 Resources

- Learn to Build a Neural Network from Scratch

# 3 Convolutional Neural Networks

Convolutional Neural Networks (CNNS) have been around for a while, but the first real breakthrough came when GPUs were discovered to greatly accelerate them. Then, the creation of AlexNet in 2012 kicked off its widespread usage in computer vision tasks. Now, there are many variants of CNNs used for many purposes ranging from image classification, object detection, segmentation, data compression, analyzing audio/time series data, etc. So how do they work? We're going to look at a "vanilla" CNN for classification, which is composed of two parts:

1. Convolution layers

2. Fully-connected neural network

The convolution layers perform feature extraction. They reduce the spatial size of the data in exchange for increased semantic depth. The FCN works the same as the aforementioned FCN, the only difference being that the input is a feature-dense representation of the data instead of being raw datapoints. Yet, if FCNs are universal function approximators, **why do we even need CNNs**? The issue is we're living in a world limited in compute power. For example, CNNs have been demonstrated to take orders of magnitude shorter time to reach similar performance compared to fine-tuned FCNs.

## 3.1   Convolution Layers

I mentioned earlier that CNNs reduce the spatial dimensions of the data in exchange for increased semantic depth. What exactly does that mean?

Each convolution layer consists of many kernels, or filters, each of which is a matrix. Let's say we have input (5, 5). Notice anything interesting? Example: a 5x5 grayscale matrix representing a "0" shape (values closer to 1 are more solid)

$$\begin{bmatrix} 0.2 & 0.8 & 0.9 & 0.8 & 0.2 \\ 0.8 & 0.1 & 0.1 & 0.1 & 0.8 \\ 0.9 & 0.1 & 0.2 & 0.1 & 0.9 \\ 0.8 & 0.1 & 0.1 & 0.1 & 0.8 \\ 0.2 & 0.8 & 0.9 & 0.8 & 0.2 \end{bmatrix}$$

The human eye is very good at recognizing patterns - we can immediately see that the values close to 1 form a 0 shape. But how is a computer supposed to grasp that concept? The key is applying KERNELS: let's see if we can figure out what this one does:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

These kernels are applied to data by taking the hadamard (element-wise) product of the original matrix with the filter followed by the sum. If we pass it over an image that's transitioning from dark to light (from top to bottom), the kernel will yield a very low value. This is because dark grayscale values are closer to zero than lighter grayscale values.

Conversely, if it's going from light to dark, the kernel will yield a very high value. If it's the same color all the way through, the kernel will yield a value close to zero. This kernel captures **transitions into and out of horizontal edges**.
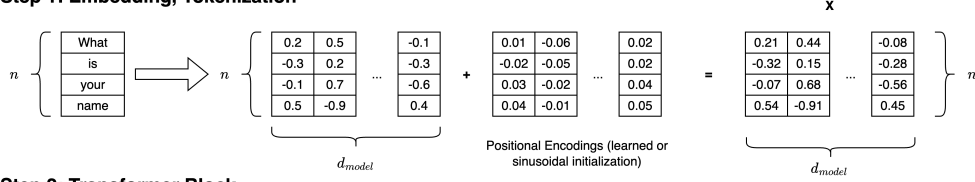
How do we find these kernels? For CNNs, they are actually randomly initialized and learned via backpropagation. In addition to these convolutional operations, pooling operations (e.g. maxpool) are often used. As with many things in deep learning, nobody knows exactly why pooling works so well but experimentally it's quite robust.
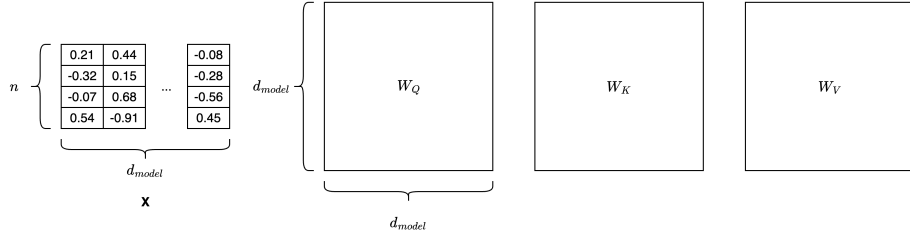
## 3.2  Resources

- CNN explained

- CNN visualizer (hands-on)
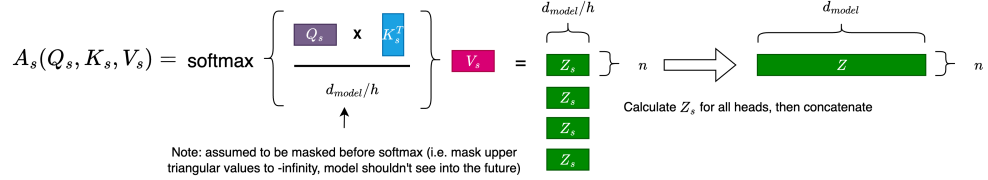
# 4 Transformers

**Step 1: Embedding, Tokenization**

| What |
| is |
| your |
| name |

$n$

| 0.2 | 0.5 | ... | -0.1 |
| -0.3 | 0.2 | | -0.3 |
| -0.1 | 0.7 | | -0.6 |
| 0.5 | -0.9 | | 0.4 |

$d_{model}$

+

| 0.01 | -0.06 | ... | 0.02 |
| -0.02 | -0.05 | | 0.02 |
| 0.03 | -0.02 | | 0.04 |
| 0.04 | -0.01 | | 0.05 |

Positional Encodings (learned or sinusoidal initialization)

=

**x**

| 0.21 | 0.44 | ... | -0.08 |
| -0.32 | 0.15 | | -0.28 |
| -0.07 | 0.68 | | -0.56 |
| 0.54 | -0.91 | | 0.45 |

$d_{model}$

**Step 2: Transformer Block**

| 0.21 | 0.44 | ... | -0.08 |
| -0.32 | 0.15 | | -0.28 |
| -0.07 | 0.68 | | -0.56 |
| 0.54 | -0.91 | | 0.45 |

$d_{model}$

**x**

$W_Q$     $W_K$     $W_V$

$d_{model}$

**a) Calculate Query, Key, Value Matrices & Divide for Multi-Head Attention ($h = 4$)**

$Q = XW_Q$       $K = XW_K$       $V = XW_V$

$Q_s$  $Q$       $K_s$  $K$       $V_s$  $V$

$d_{model}/h$
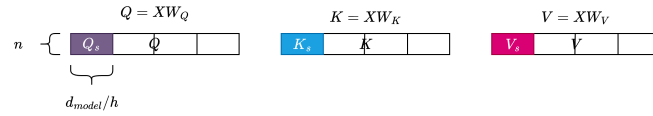
**b) Compute Attention (repeat for all sub-query/key/value) & concatenate**

$$A_s(Q_s, K_s, V_s) = \text{softmax}\left\{ \frac{Q_s \times K_s^T}{d_{model}/h} \right\} V_s = Z_s$$

$Z_s$ $Z_s$ $Z_s$ $Z_s$

$d_{model}/h$

Calculate $Z_s$ for all heads, then concatenate

$Z$

$d_{model}$

Note: assumed to be masked before softmax (i.e. mask upper triangular values to -infinity, model shouldn't see into the future)

**c) Apply learned matrix to mix info across heads**

$$O = ZW_o$$

$O$  =  $Z$  x  $W_o$

$d_{model}$     $d_{model}$     $d_{model}$

**d) Apply layernorm + residual connections**

This stabilizes training & reduces overfitting.

$$X' = LayerNorm(O + X)$$

**e) Apply MLP head (introduce nonlinearity + understanding)**

$$H = MLP(x) = \sigma(W_2\sigma(W_1 x^T + b_1) + b_2)$$

$W_1$ projects the input from $(n, d_{model}) \to (n, d_{mlp})$, where $d_{mlp} \gg d_{model}$. This enables the extracting of more complex/meaningful ideas from individual embeddings. $W_2$ projects back down to $(n, d_{model})$

**Step 3: Output/Unembedding**

$Logits = HW^{ET} \in \mathbb{R}^{n \times v}$, where $W^E \in \mathbb{R}^{v \times d_{model}}$ and is the original embedding matrix with $v$ = vocabulary size

Take the softmax across the first dimension (rows), and you'll get probabilities summing to 1 for each token across each row! Depending on training/inference, you may sample the final token only (and get something like "my") or sample all tokens.

**Note: this doesn't cover KV-cache or any additional features of the transformer (e.g. dropout) - just the bare necessities!**

Figure 1: Transformer overview. Figure made by author. Zoom in to read better!

## 4.1 Overview

Transformers are composed of three parts:

1. Encoding

2. Transformer Blocks (Multi-Head Attention + MLP)

3. Output

Thinking about this from a high-level overview, in the case of a model like GPT, what might happen is as follows. Take some sentence composed of n words and embed those words using some algorithm into a higher dimensional space (n x $d_{model}$). Say that GPT has a vocabulary size of $v$. This embedding algorithm basically maps a token within that vocabulary to a new vector in $\mathbb{R}^{d_{model}}$.

The goal for each attention layer is to essentially "reweight" each of the $n$ embeddings based on information from other embeddings. At the end of the algorithm, the "reweighted" embeddings get "unembedded". The goal is to get a probability distribution over all possible tokens in the vocabulary for the next token after the current embedding. That can be done by taking the softmax over all the tokens in the vocabulary corresponding to each embedding.

Here is where the training/inference procedure differs. If it's training, calculate the cross-entropy loss between the ground truth next token and the predicted token for EVERY word in the sentence. If it's inference, just extract the most probable next token from the LAST word in the sentence.

Notably, some transformer models have encoders AND decoders, whereas others are only encoders (e.g. BERT) and others are only decoders (e.g. GPT). They differ in use cases, namely that encoder-based models are good for capturing deep semantic information whereas decoder-based models can generate new, realistic output. However, the difference between encoder-only and decoder-only models has been blurring a bit as models from both sides improve their understanding and generation capabilities.

## 4.2 Embedding

Coming soon!

## 4.3 Transformer Block

Coming soon!

## 4.4 Output

Coming soon!

## 4.5 Resources

- Attention Is All You Need (Transformer paper)

- PyTorch transformer implementation

# 5 Training Models

## 5.1 LMs: Reinforcement Learning with Human Feedback (RLHF)

Given some language model (LM) that's been pretrained on a large corpus of data (e.g. internet), we don't know how **reliable** or **aligned** that data is with human values. Thus, we want to **guide** the LM to incorporate basic human values, such as honesty, harmlessness, etc. Here's a high level overview of the process:

1. **Reward Model**: procure a dataset of good/bad LM outputs relative to a prompt (via human annotators), train a reward model to maximize the difference in reward between good/bad prompt. Freeze the reward model after training.

2. **KL Divergence Calculation**: for any given prompt, calculate the KL divergence between frozen and unfrozen version of LM and subtract from the reward model output.

3. **PPO gradient update**: update weights of the unfrozen LM, and repeat step 2 and 3 until satisfactory performance is achieved.
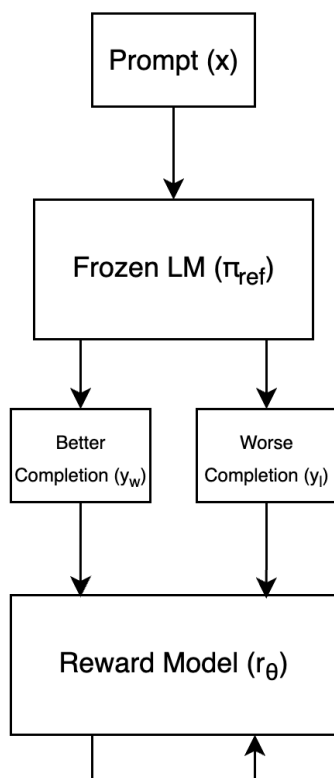
### 5.1.1 Reward Model

We want to first train a reward model that models human preference. Given a pretrained reference model $\pi_{\text{ref}}$, for each prompt $x^{(i)}$ where $i = 1, \ldots, N$, generate two outputs $y_w^{(i)}$ and $y_l^{(i)}$ such that:

$$y_w^{(i)}, y_l^{(i)} \sim \pi_{\text{ref}}(\cdot \mid x^{(i)})$$

where $y_w^{(i)}$ is the preferred (higher-quality) completion and $y_l^{(i)}$ is the less preferred (lower-quality) completion for prompt $x^{(i)}$. This yields a dataset of $(y_w^{(i)}, y_l^{(i)}, x^{(i)})_{i=1}^N$.

Step 1: Procuring labeled preference
dataset + training reward model

Step 2: Training LM

Prompt (x)

Prompt (x)

Frozen LM ($\pi_{ref}$)

$\pi_{ref}$

Fine-tuned
LM ($\pi_\theta$)

find $\pi_{ref}$(y | x) -
teacher forcing

Generate new
completion y

Update via
PPO

Better
Completion ($y_w$)

Worse
Completion ($y_l$)

$$KL = \beta \log(\frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)})$$

Reward Model ($r_\theta$)

$$L_{final} = r_\theta(x, y) - KL$$

Minimize:

$$-\log(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l)))$$
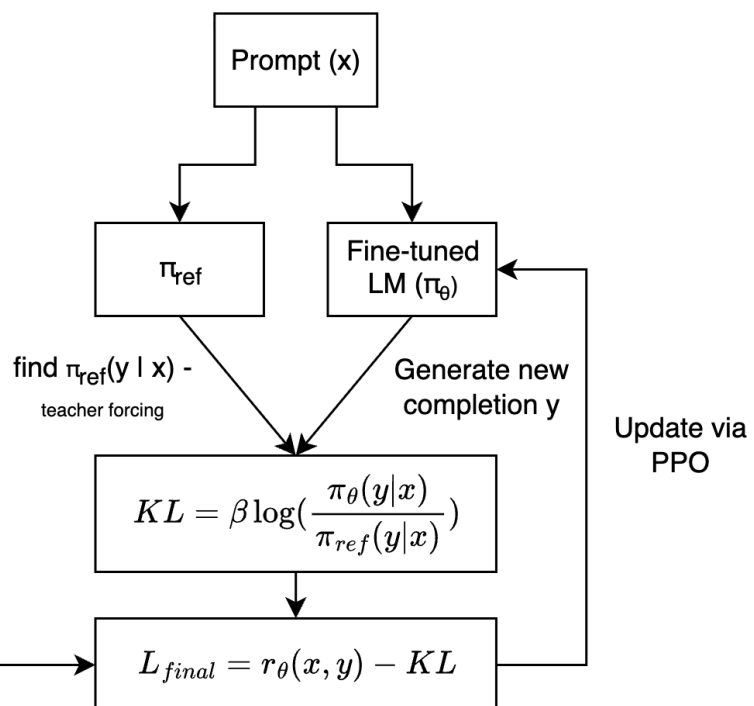
Figure 2: RLHF overview. Figure made by author.

Next, use a reward model $r_\phi$ (perhaps just a transformer an added linear layer at the end) to process each sample in the dataset, minimizing the loss

$$L_{reward} = -\log\left(\sigma\left(r_\phi(y_w) - r_\phi(y_l)\right)\right)$$

. This encourages the maximization of the reward for the good response while minimizing the reward for the bad response.

### 5.1.2 KL Divergence Calculation

Now, we have a trained reward model. Let's freeze the weights of the original LM ($\pi_{ref}$) make a trainable copy of the original LM and call it $\pi_\phi$ (this will be the final fine-tuned LM after the entire process is complete).

We need to prevent $\pi_\phi$ from "reward hacking" (i.e. giving nonsensical outputs to fool the reward model), so we calculate KL divergence between outputs of $\pi_\phi$ and $\pi_{ref}$. This is given as $KL = \beta \log\left(\frac{\pi_\phi(y|x)}{\pi_{ref}(y|x)}\right)$.

## 5.2 LMs: Direct Preference Optimization (DPO)
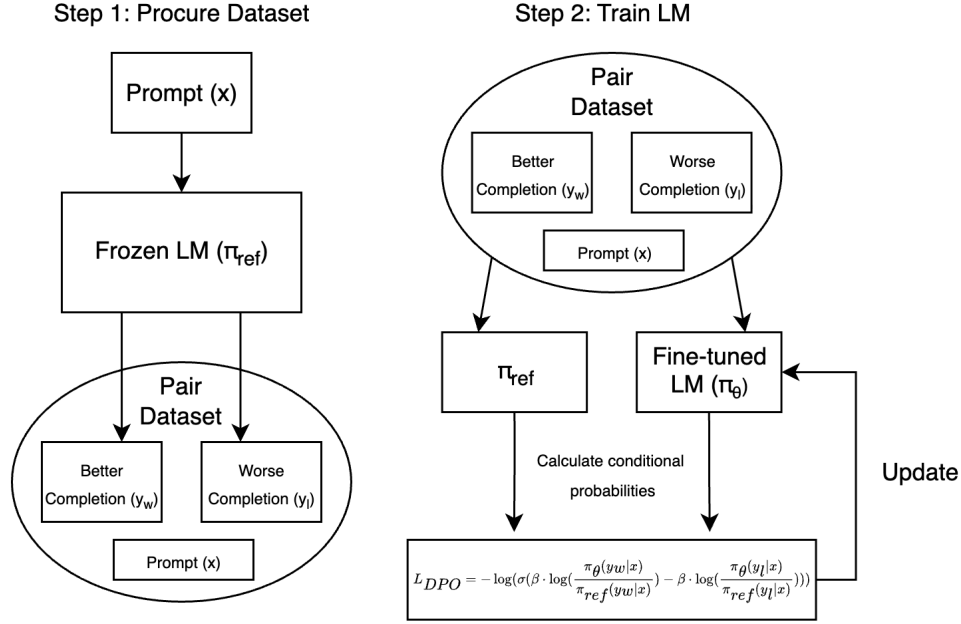


Figure 3: DPO overview. Figure made by author.

DPO eliminates the need for training an explicit reward model. The loss is formatted as $L_{DPO} = \log(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l)))$. The goal is to reparameterize the reward function $r_\theta$ such that it's in terms of the original LM outputs (thus removing the need to train a separate reward model). Specifically, $r_\theta(x, y)$ is reparameterized as $\beta \log(\frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}) + \beta Z(x)$. Briefly, $Z(x)$ is a partition function that represents some normalization factor over all possible outputs for the LM. For more information on the mathematical justification and details, visit the paper. Because we take the difference between $r_\theta(x, y_w)$ and $r_\theta(x, y_l)$, the $Z(x)$ term cancels out, which saves us a lot of time! Thus, the final loss is represented as:

$$L_{DPO} = -\log(\sigma(\beta \cdot \log(\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)}) + \beta \cdot \log(\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)})))$$

12

The authors argue that the intuition for why this approach works so well is because LMs can natively encode reward (hence "your language model is secretly a reward model"). Notably, DPO relies assumes the validity of the Bradley-Terry model. This states that human preference can be modeled by scalar values and pairwise comparisons are valid methods of determining the best model output. In reality, human preference is often not transitive (i.e. preferring A over B and preferring B over C does not always guarantee A is better than C). This is also a fundamental limitation for the normal RLHF setup.

## 5.3   Resources

- Training language models to follow instructions with human feedback (RLHF paper)

- Direct Preference Optimization: Your Language Model is Secretly a Reward Model