(DRAFT) BitTensor: Biological Scale Neural Networks

July 2019

Abstract

BitTensor allows a new class of Machine Learning model which trains across a peer-to-peer network. It enables any computer and any engineer in the world to contribute to its training. The nature of trust-less computing necessitates that these contributions are driven by incentives rather than by direct control from any one computer. We use a digital token to carry that incentive signal through the network. The magnitude of this incentive is derived from a p2p collaborative filtering technique similar to Google's Page Rank algorithm. As a network product we focus on learning unsupervised multi-task representations, starting from language and extending the network to image and speech. The result is a sufficiently general product which is useful to a large number of downstream stake holders. The lack of centralization allows the structure to grow to an arbitrary size across the internet. Both the cost and control of the system is distributed and the network's informational product is priced into the reward token's value.

1 Introduction

It is almost always the case that you can improve the performance of a machine intelligence system by leveraging more computation.[16][32] It makes sense then to seek a method of combining the largest connected source of computation available to man: the internet.

We take inspiration from Bitcoin [33], the largest super-computer in existence, an open-source decentralized software system which reaches every corner of the web, and combined through incentive, is multiple times larger than all compute resources amassed by Google – consuming more energy than Austria.

2 Spikes

An Artificial Neural Network (ANN) decomposes into a network of smaller components where each component can be described as a function $f(x)$ with parameters θ, which acts on inputs x, (of some type, text, image, etc) to produce outputs $y = f(x)$. The i^{th} of these component functions is a composition of downstream components $d_0, ..., d_j, ..., d_n$, such that:

$$f_i = f_i(x, d_0(x), ..., d_j(x), ..., d_n(x)) \qquad (1)$$

and, by reflection, be one compositional part to a set of upstream components, $u_0, ..., u_j, ..., u_n$, where:

$$u_0, ..., u_j, ..., u_n = u_j(x, ..., f_i(x), ...) \qquad (2)$$

This structure is recursive and for some input (x), we can attain the network output at one component $f_i(x)$, by propagating the input downstream and then reducing the signal back up. For instance, in a normal Feed-forward Neural Network we would begin at the loss function, propagate to the input layer, and then back up. If there are graph-cycles, and this may be the case in an ad-hoc network over the web, we cut recursion when messages pass through the same component for the second time.

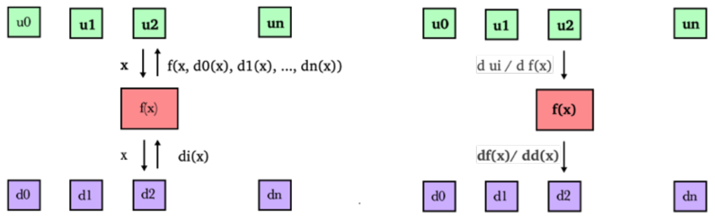
3 Gradients

We assume each component is differentiable, and supports reverse-mode automatic differentiation [36] needed for back-propogation learning. In this framework, the i^{th} component accepts an error signal of form $(x, \partial y_i)$, with $\partial y_i = \frac{\partial u_j}{\partial y_i}$ and sends error signals downstream of form $\frac{\partial u_j}{\partial dj} = \frac{\partial u_j}{\partial y_i}\frac{\partial y_i}{\partial d_j}$ for each downstream neighbor d_j.

The design is no different than the gradient calculations in the differential graph architecturs built with TensorFlow or PyTorch [34][35]. Where the error signals advise the i^{th} component on training its parameters θ to minimize the loss defined by an upstream component u_j, namely L_{uj}. The accumulation of these error signals translate into a gradient batch $[\Delta\theta_1, \Delta\theta_2 ... \Delta\theta_N]$ where each may be applied to minimize the respective upstream losses.

During training each component is applying gradients using Stochastic Gradient Decent (SGD), where at each training-step the i^{th} component averages the batch to determine the accumulated step. We extend this slightly by first normalizing gradients from within component u_j to find a batch $[\Delta\theta_0^*, ..., \Delta\theta_j^*, ..., \Delta\theta_n^*]$, and then finally across each component u_j according to a set of weights $[w_0, ..., w_j, ..., w_n]$. Our full step is thus a weighted average of error signals from each upstream component:

$$\Delta\theta = \frac{1}{\sum_{j=1}^{N} w_j}(\Delta\theta_1^* * w_0 + ... + \Delta\theta_j^* * w_j + ... + \Delta\theta_n^* * w_n) \qquad (3)$$



Note, for brevity's sake, we define upstream as the direction information accumulates and downstream as the direction gradients propagate. Calls on the network where information accumulates are called *Spikes* and calls propagating gradient information downstream are called *Grads*.

4 Incentives

We are extending previous work in Neural Network training by moving the training process from a datacenter into a decentralized computing domain across the internet, the computing domain we are in is trust-less, no computer is privileged[37], there is no single user of the network, and some computers may be incompetent, offline, or malicious. Because of these constraints, we must use incentive to draw our compute nodes into line. That incentive should drive them to stay active and to train in alignment with a network objective.

To begin, we define that network objective as a sum over the local objectives defined at each node. More, in order to bind the concept of value into the network objective and give us some protection against Sybil attacks, we scale each local objective by a stake vector S. Then for S_i, the stake attached to the i^{th} loss, L_i we have:

$$\text{network objective} = N^* = min \sum_i^N S_i * L_i \qquad (4)$$

As a form of incentive we wish to mint new stakeable tokens to components in-proportion to their contribution optimizing our network objective. We do this by asking what it would cost, in terms of loss, to prune each node from the network. With ΔL_{ij} the pair-wise contribution score defined as the change to the i^{th} loss w.r.t removal of the j^{th} component:

$$\Delta N_j^* = N^* - \sum_i^N S_i * \Delta L_{ij} \qquad (5)$$

ΔN_j^* is our contribution score for the j^{th} component.

5 Contribution

The pair-wise contribution score ΔL_{ij} can be attained using a 2nd order approximation of the i^{th} loss, L_i with respect to a change Δd_j reflecting the removal of that peer from the network. Namely,

$$\Delta L_{ij} = L_i(d_j + \Delta d_j) - L_i(d_j) \approx g^T \cdot \Delta d_j + \frac{1}{2}\Delta d_j^T \cdot H \cdot \Delta d_j \qquad (6)$$

$$g_x = (\frac{\delta L_i(x)}{\delta d_j}) \qquad (7)$$

$$H \approx E_X[g_x * g_x] \qquad (8)$$

This equation (6) evaluates at some $x \in X$, such that g_x (7) is the gradient at that point. The remaining term is the Hessian, and can be viewed as an empirical estimate of the Fisher information [17] of our input d_j. This term can be approximated using an expectation over our a set of examples X. And our full pair-wise contribution score can be calculated as follows:

$$\Delta L_{ij} \approx \sum_{x \in X} g_x \cdot \Delta d_j + \frac{1}{2N}\Delta d_j \sum_{x \in X} g_x * g_x \qquad (9)$$

Where the $*$ in (8) and (9) represents element-wise multiplication of g_n, such that H is an $n \times 1$ vector, where each element is the Fisher Information of a single dimension in d_j, and both terms collapse to a scalar after the dot product with Δd_j.

For directly connected components, this approximation (9) is available during the backward pass of computing the components's gradient and the contribution signal can therefore be found at little extra computational cost. Note, various other forms of contribution signals (a.k.a salience methods) can be used, for instance here [27][29][30]

For non-directly connected components, i.e. i and k with $i -/-> k$, ΔL_{ik} is not immediately derivable during the backward pass of computing the gradient. Instead, we must derive these non-direct paths by applying the chain rule to (6) to find the following transitive relation:

$$\text{Given } \Delta L_{ij} \text{ and } \Delta L_{jk}$$

$$\Delta L_{ik} = \Delta L_{ij} * \Delta L_{jk} \qquad (10)$$

Which is intuitive, following immediately from the notion of transitive contribution: If a component i contributes to component j, it should multiplicatively contribute to the components using j since they are compositions of its parent. This transitive quality is also exploited in [29] for computing neural importance weights.

In order to evaluate (5), namely ΔN_j^*, the global contribution score for component j, we need an approximation for every ΔL_{ij} pair. We can attain this as a corollary of (10) by progressively computing every path through the network. A common method is the Power Iteration over an adjacency matrix, G, where each edge weight is set to the local ΔL_{ij} score computed by the directly connected components i and j.

Our Power Iteration takes a modified form of the following recurrence relation,

$$\Delta N^*(t+1) = G * \Delta N^*(t)$$

Algorithm 1 Modified Eigen Trust calculation of contribution vector ΔN^*

```
Require: $S$ = Stake  $[N \times 1]$ vector
Require: $\Delta Lii$ = Contribution from i to i  $[N \times 1]$ vector
Require: $\Delta Lij$ = Contribution from i to j  $[N \times N]$ matrix
  $\Delta N^* \leftarrow S \cdot \Delta Lii$
  $temp \leftarrow \Delta Lij * S$
  while $temp > \epsilon$ do
    $\Delta N^* \leftarrow \Delta N^* + (temp \cdot \Delta Lii)$
    $temp \leftarrow \Delta Lij * temp$
  end while
  $\Delta N^* \leftarrow \frac{\Delta N^*}{sum \Delta N^*}$
  return $\Delta N^*$
```

This is similar to the EigenTrust algorithm [23] or Google Page Rank [1], but with local-contribution scores instead of recommendations or web links.

6 Emission

A finite quantity of stake, s_j is being allocated to each loss term in the network. This is combined with local ΔL_{ij} calculations and the global contribution calculation above, to produces a score for each node ΔN_j^*. We emit new tokens within the graph to components in proportion to their contribution. As follows:

The emission algorithm above runs at discrete time-steps to produce a stream of newly minted tokens for each component in the network. In order to ensure

Algorithm 2 Calculate Emission vector ΔS

```
Require: $S$ = Stake vector
Require: $\Delta Lii$ = Self-attribution vector
Require: $\Delta Lij$ = Local-attribution matrix
Require: $\sigma$ = tokens released per block.
  loop
    $\Delta S = get\Delta L(S, \Delta Lii, \Delta Lij)$
    $\Delta S = \Delta S * \sigma$
    $S = S + \Delta S$
  end loop
```

that the emission scheme is fair and deterministic we use a Blockchain consensus engine to perform its calculation. With this technology the specifics of token emission stay fixed and the local attributions scores ΔLij and ΔLii are posted globally such that every component can determine how they are attaining token emission.

In practice the emission calculation is too expensive to carry out all at once (given the need for global consensus) we remedy this by amortizing the cost across multiple calls, where each call computes a portion of the emission calculation.

As an aside, we note that this form of local contribution and emission scheme may be similar in nature to the propagation of the neurotrophic factor BDNF in the brain.[2]

7 Market

The magnitude of the token-stream for the i^{th} component is t_i which is directly related to the contribution scores ΔLij used by the emission calculation. However, the true value of these scores are subjective: computed at each component in an unauditable and trustless manner. As such, there is no-guarantee that contribution scores ΔL_{ij} are posted or advertised correctly by any component. What keeps these values honest?

In what follows we explore the competitive nature of the network as a whole with an analysis of utility at each component and show how there is a dominant strategy for each component where contribution scores are the market is at least 75% efficient.

To begin, each node has some cost c_i associated with running as a member of the network (most likely compute and network costs). Further more, each component defines a loss L_i and utility function associated with optimizing its local loss: $U(-L_i)$. This function is continuous and strictly increasing in its loss term. We assume the utility function and cost term are measured in the network token and thus comparable to t_i.

$$\text{ith utility} = u_i = U(-L_i) \ + \ t_i \ - \ c_i \qquad (11)$$

For simplicity we split our analysis into two scenarios. In the first, we consider a component who is solely attempting to maximize the utility attained by minimizing its loss function $U(-L_i)$, it has no upstream peers. And in the second, we consider a component who is solely attempting to maximize t_i, it has no downstream peers.

In the first scenario, our component i decides how it will score the downstream component j. j's strategy is fixed, and is competitively applying gradients to its parameters according to the method described in (3) with the weight attached to the i^{th} upstream channel's gradients equal to the score L_{ij} bid by the component i. Namely,

$$\Delta\theta = \frac{1}{\sum_{i=1}^{N} L_{ij}} (... + \Delta\theta_i^* * \Delta L_{ij} + ...) \qquad (12)$$

Where the change in L_i given the change to j's parameters, $\Delta\theta$, can be found using the same second order approximation used before:

$$\Delta L_i = L_i(\theta + \Delta\theta) - L_i(\theta) \approx g^T \cdot \Delta\theta + \frac{1}{2}\Delta\theta^T \cdot H \cdot \Delta\theta \qquad (13)$$

Note that from (12) and (13) that i's loss utility $U(-L_i)$ is concave, strictly increasing, and continuous in the bid ΔL_{ij}. This is a proportional allocation game where i is bidding against j's other upstream peers for each SGD step applied to its parameters. We know from [32] that there exists a unique Nash equilibrium and it is an optimally solution which maximizes the payoff of each upstream peer. However this solution is not necessarily found since the bidders in this system are $[u_0, ..., u_j, ...u_n]$, price anticipating (understand their effects and the bids of the other participants). Regardless, we are given from [32] that the competitive equilibrium has an efficiency loss which is exactly 25% in the worst case. The proof of this is given in [32:22.7]

In the second scenario, the downstream component chooses a market clearing strategy i.e the choice of weights attached to (3) which decide how the weighted gradient step is applied. This is a market clearing strategy in a proportional allocation game where market participants $[u_0...u_n]$ are price anticipating. We know from [32] that the strategy for minimizing the worst case efficiency loss is the proportional allocation mechanism assumed above. More, we can see from the attribution calculation (9) and emission system (10) that component i maximizes its token stream t_i, when its output is most informationally significant to its upstream peers. In other words, it maximizes utility by learning a representation which maximizes each contribution score ΔL_{ij}) weighted by stake.

Note, scenarios which include a combination of the two, for instance a component which is both upstream and downstream, should apply a weighted combination of both dominant strategies.

8 Representation Learning

The above Neural Network architecture and incentive system requires a Machine Intelligence problem which is general enough to interest a diverse set of stake holders. The problem should be sufficiently difficult to warrant a collaborative approach and the data used to train it should be ubiquitous and cheap.

For our purposes, we choose unsupervised representation learning [5, 6, 7, 9, 10, 14, 24, 25], where the network trains on large-scale unlabeled corpora to learn a feature basis ('representation') of inputs. These representations are a form of product, which uniquely identifies and disentangles the underlying explanatory factors of an input (a.k.a an inductive bias). This is a widely used product and arguably a fundamental task in the development of an AI which understands the world around it. [26]

We initially focus on Language Representation from text, where components build an understanding of natural language and will respond to queries in pure unicode strings with a vector representation. For the sake of generality, we leave tokenization and parsing to each component and limit outputs across the network to fixed length vectors.

"raw natural language text" $---> [f(T)]--->$ [fixed length representation]

The standard scheme for learning a representation is as follows. First, the raw text is tokenized, for instance at the word[32], sentence[31], or byte level[7], to create a sequence of discrete tokens $T = (t_1, t_2, ..., t_n)$. The modeling task consists in learning a representation for that sequence $f(T) = f(t_1, t_2, ..., t_n)$.

Our representation function can be trained in any number of supervised or unsupervised forms. However, commonly this is achieved in an unsupervised manner, where a Neural Network architecture parameterized by θ, trains $f(T)$ to condition the probability of tokens T' in its near context.

$$maximize \prod_{T' \in M} P(T'|f(T))$$

Where M is some unlabeled dataset of text, of which there are many freely available. This high-level paradigm has been shared and successfully applied by a large number of models to build incredibly powerful representations for language. These include:

- ELMO [31] representations from bidirectional recurrent neural networks.
- BERT [5] representations from multi-word masking strategies.
- MT-DNN [14] representations from multi-task knowledge transfer.
- GPT-2 [7] representations from a question-answering dataset.
- XLM representations from cross-lingual datasets.
- ERNIE [10] representations from entity/phrase level masking.
- XLNet [9] representations across all mask permutations.

Interestingly, the idea of learning representations in context was nicely described by John Rupert Firth in 1957 with the following quip: "you shall know a word by the company it keeps". It could be argued that the human mind is constantly attempting to know the world by the company it keeps, namely, by constructing a representation of the world which allows us to predict the future from the present – a useful task for any organism.

9 Experiments

We begin by running a trainable Neural Network across multiple distributed computers. We pre-load the network with the GoogleUSE model and ELMO, and attach multiple additional nodes which act by connecting and learning off these model outputs. We combine this system with two other node types which query the network in attempt to learn Word Embeddings with low perplexity score.

Each node is ranking itself and its neighbors using the attribution method described above. These are being sunk to a decentralized consensus machine running on EOS. The contract currently runs in Testnet but can be easily loaded onto the live network. We apply the Emission Scheme explained above to mint tokens through the network.

The code for these experiments is open source. github.com/unconst/Bittensor TODO

10 References

1 The PageRank Citation Ranking
http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

2 Brain-derived neurotrophic factor and its clinical implications
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4697050/

3 Attention is all you need
https://arxiv.org/abs/1706.03762

4 Universal Language Model Fine-Tuning for Text Classification
https://arxiv.org/abs/1801.06146

5 Bi-directional Encoder Representations from Transformers
https://arxiv.org/abs/1810.04805

6 Googles Transformer-XL
https://arxiv.org/abs/1901.02860

7 Open AI GPT2
https://openai.com/blog/better-language-models/

9 XLNet
https://arxiv.org/abs/1906.08237

10 ERNIE: Enhanced Representation through Knowledge Integration
https://arxiv.org/abs/1904.09223

11 RoBerta: A Robustly Optimized Bert Pre-training Approach
https://arxiv.org/abs/1907.11692

12 Cross-lingual Language Model Pre-training
https://arxiv.org/pdf/1901.07291.pdf

13 Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer
https://arxiv.org/abs/1701.06538

14 One Model To Learn Them All
https://arxiv.org/abs/1706.05137

15 AHaH Computing–From Metastable Switches to Attractors to Machine Learning
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0085175

16 Distilling the Knowledge in a Neural Network
https://www.cs.toronto.edu/ hinton/absps/distillation.pdf

17 Faster Gaze Prediction With Dense Networks and Fisher Pruning
https://arxiv.org/pdf/1801.05787.pdf

18 Overcoming catastrophic forgetting in neural networks
https://arxiv.org/abs/1612.00796

19 Bitcoin: A Peer-to-Peer Electronic Cash System
https://bitcoin.org/bitcoin.pdf

20 IPFS - Content Addressed, Versioned, P2P File System
https://arxiv.org/abs/1407.3561

21 Self-Attention with Relative Position Representations
https://arxiv.org/pdf/1803.02155.pdf

22 Generating Wikipedia by Summarizing long sequences
https://arxiv.org/pdf/1801.10198.pdf

23 The EigenTrust Algorithm for Reputation Management in P2P Networks
http://ilpubs.stanford.edu:8090/562/1/2002-56.pdf

24 Distributed Representations of Words and Phrases and their Compositionality
https://arxiv.org/pdf/1310.4546.pdf

25 Skip-Thought Vectors
https://arxiv.org/abs/1506.06726

26 Representation Learning: A Review and New Perspectives
https://arxiv.org/pdf/1206.5538.pdf

27 Optimal Brain Damage
http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf

28 A Hierarchical Multi-task Approach for Learning Embeddings from Semantic Tasks
https://arxiv.org/abs/1811.06031

29 NISP: Pruning Networks using Neuron Importance Score Propagation
https://arxiv.org/pdf/1711.05908.pdf

30 Overcoming catastrophic forgetting in neural networks
https://arxiv.org/abs/1612.00796

31 Deep contextualized word representations
https://arxiv.org/pdf/1802.05365.pdf

32 Efficient Estimation of Word Representations in Vector Space
https://arxiv.org/abs/1301.3781

33 Algorithmic Game Theory - CMU Computer Science
https://www.cs.cmu.edu/ sandholm/cs15-892F13/algorithmic-game-theory.pdf

34 The Bitter Lesson
http://incompleteideas.net/IncIdeas/BitterLesson.html

35 Bitcoin: A Peer-to-Peer Electronic Cash System
https://bitcoin.org/bitcoin.pdf

36 Large-Scale Machine Learning on Heterogeneous Distributed Systems
http://download.tensorflow.org/paper/whitepaper2015.pdf

37 Automatic differentiation in PyTorch
https://openreview.net/pdf?id=BJJsrmfCZ

38 Compiling Fast Partial Derivatives of Functions Given by Algorithms
https://www.osti.gov/biblio/5254402-compiling-fast-partial-derivatives-functions-given-algorithms

39 IPFS - Content Addressed, Versioned, P2P File System
https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf