

Useful Proof of Work (Solving Optimization Problems)

In order for proof of work to become a viable solution as a consensus algorithm, the work that is done by miners needs to have some inherent use to incentivize its use.

Bitcoin has miners solve an arbitrary mathematical problem that only has the purpose of proving a user's computational strength as a measure of commitment to the network. If the puzzle that miners were solving had some inherent value other than showing off a miner's raw computational power, then that block chain would have much more value and practical use.

There have been coins in the past that have attempted to solve exactly this issue of useless computation, like Primecoin whose proof of work algorithm would have miners generate prime numbers according to a specialized algorithm. This POW algorithm has inherent use because discovering new primes has applications other than proving work to the blockchain network. However people would probably agree that only a small portion of the population would find new primes inherently valuable to them.

What this paper proposes is that miners solve user defined puzzles in the form of finding optimal solutions to neural networks. For example a user wants to solve the classic beginner machine learning problem – the XOR logic gate. The user specifies the desired meta data for the optimization problem, i.e. supervised learning, number of layers, fully connected layers, as well as the training data (only in the case of supervised learning). The user signs the network meta data and transaction with their wallet address and if the funds in the wallet are equal or greater to the cost of running the network with the specified metadata and associated training data then miners attempt to optimize the network. Users will also be able to generate the data from equations running in turing complete environments. For example in the case

of a reinforcement learning model that relies on interactions with a simulated environment and not labeled inputs and outputs. The equations defining the environment would be the reward function for the network. Additionally there can be a built in feature that allows users to reference and call upon other previously optimized networks further back in the block chain allowing users to build on each other's work and assuring that complicated models are only trained once and can be referenced by everyone later.

```

1  {
2    "name": "My first neural net on the Legatus blockchain!",
3    "type": "0001",
4    "meta": {
5      "layers": [2, 2, 1],
6      "training": "supervised",
7      "validation": "training"
8    },
9    "training_data": [
10     [[0,0], [0,1], [1,0], [1.1]],
11     [[0], [1], [1], [0]]
12   ],
13   "max_error": 0.002,
14   "priority_multiplier": 3,
15   "expiration_time": 33721,
16   "transaction": {
17     "job_hash": "4b4897ebe...",
18     "from": "8979eb90abe...",
19     "amount": 1900,
20     "nonce": "87b9018828ea00...",
21     "current_block": 20000,
22     "signature": "EBE80109EEE..."
23   }
24 }
25
26

```

meta info that will define how the solution to this problem is to be formatted

for this problem a labeled set of training data has been provided

defines the priority of this job (higher priority deployments cost more LGTS)

standard transaction info to make sure this is the owner of this account


Example JSON object that will deploy a new neural network to be solved by miners for rewards.

An obvious problem that would arise particularly at the beginning of the network's lifetime is not enough puzzles being posted by users to solve. How would the network generate blocks if all the posted jobs (these are the user defined optimization problems) have been adequately (defined in the user's metadata when uploading their network) solved? The solution is a community optimization problem deterministically generated from the hash of the previous block that requires the *same operations* to solve as the user submitted optimization problems would. This would incentivize miners to hyper optimize their hardware and algorithms for problems like: how to most efficiently run the error backpropagation algorithm (commonly used for

training neural networks) for a multi-layered neural network with a sigmoid activation function.

This “community neural network” defines the block problem and would be analogous to the bitcoin network finding the next hash that fulfills the network’s current difficulty:

Block 672907 ⓘ

Hash	00000000000000000006d62d68aa7b8a13135305a99b76f2a7af7db4fad598a5 
Confirmations	1
Timestamp	2021-03-02 19:39
Height	672907
Miner	Unknown

For the Legatus network to generate a problem in the form of a neural network, the associated training data must be random in its values, but still solvable by the parameters of the network determined by the current block difficulty (which is determined similarly to how bitcoin determines how many 0's it needs in front of a hash for other nodes to accept it).

The network and its hyperparameters are defined by the block difficulty (the positive integer value), and is treated as a classifying problem with its inputs being deterministically generated from the last block hash so all miners have the same data to train on. The solution is then presented by a miner in the form of a byte string representing all the weights and biases defining the network for other nodes to validate and accept the miner's new block.

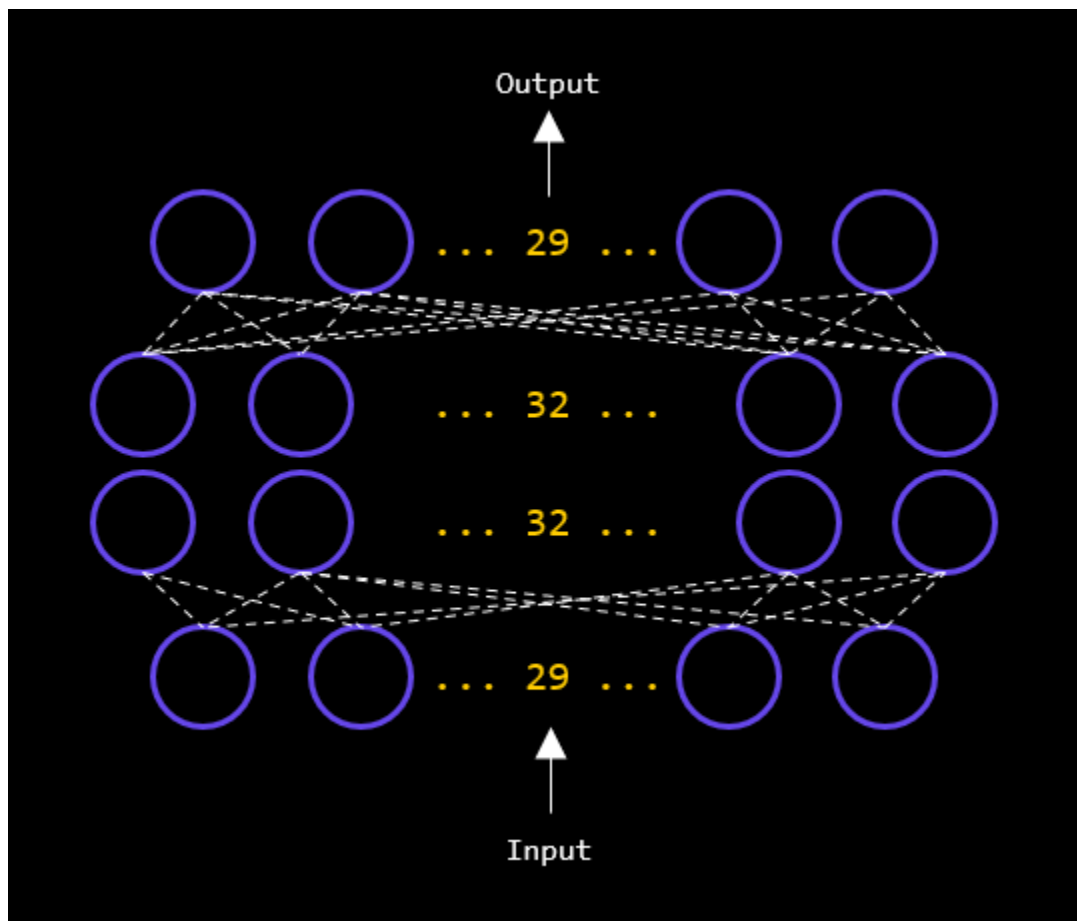
Example:

Block difficulty: 33

The training data and network architecture is generated from this equation:

```
416 //Step 1 - the complexity of the problem is defined by this number
417 let blockDifficulty = 33;
418
419 //Step 2 - the architecture of the neural network is generated
420 let numberOfInputs = blockDifficulty;
421 let numberOfOutputs = numberOfInputs;
422 let numberOfHiddenLayers =
423     Math.floor(Math.log10(blockDifficulty)) + 1;
424
425 //Step 3 - the training/validation data scales with the blockDifficulty variable
426 let numberOfDataPoints =
427     Math.floor(numberOfOutputs*((numberOfHiddenLayers)*0.9));
428
429 //Step 4 - last parameter for the neural network
430 let hiddenLayerSize = Math.floor(numberOfInputs*1.1);
431
```

This produces a network schema of: [33, 36, 36, 33]



This block's training data:

```
Training Data [0]
```

```
Input:
```

```
[0.5405415033568339, 0.2146727822920802, ... 29 ..., 0.5406370098810445,  
0.3074915706903736]
```

```
Output:
```

```
[1, 0, ...29..., 0, 0]
```

```
... numberOfDataPoints - 2 ...
```

```
Training Data [numberOfDataPoints-1]
```

```
Input:
```

```
[0.25801994585184773, 0.5406370098810445, ... 29 ..., 0.3074915706903736,  
0.9526428061937375]
```

```
Output:
```

```
[0, 0, ...29..., 0, 1]
```

There will always need to be a greater number of data points than output neurons for this classifier problem because there needs to be overlap in the data so that miners can't use tricks like setting entire layers of weights to 1 and only train the necessary layers to solve the puzzle faster – however it's also important that the network can guarantee that an error under **5%** is *possible* for the network's architecture otherwise the entire blockchain could stall because the most recent block hash defines an unsolvable network.

Once the network of the specified architecture is trained to a point where its error is less than **5%**, this puzzle is considered solved and is submitted to the network for other nodes to verify.

The outcome of this algorithm is a trustless blockchain network that is built to thrive in the age of artificial intelligence as miners aim to build hyper

optimized techniques for solving useless neural networks, so that user submitted useful networks can be solved just as efficiently making this “useful work”.

It will be up to the initial founders of this coin to release a miner that can simply run the protocol, and as interest and competition increases more optimized algorithms will appear from the community.

This is preliminary data obtained from a single core CPU comparing solve times relative to block difficulty using the technique outlined above.

Data

Max Error Threshold	Block Difficulty	Complexity (combined # of weights + biases that need to be tuned)	Learning Rate	Time to Solve (ms)
5%	1	~	~	
5%	2	~	~	
5%	3	~	~	
5%	4	~	~	
5%	5	65	0.13	68
5%	6	90	0.13	45
5%	7	119	0.13	33
5%	8	152	0.13	56
5%	9	189	0.13	57
5%	10	383	0.13	502
5%	11	454	0.13	312
5%	12	531	0.13	391
5%	13	614	0.13	464
5%	14	703	0.13	550
5%	15	798	0.13	523

5%	16	899	0.13	737
5%	17	1,006	0.13	702
5%	18	1,119	0.13	995
5%	19	1,238	0.13	948
5%	20	1,448	0.13	821
5%	21	1,583	0.13	1,210
5%	22	1,724	0.13	1,339
5%	23	1,871	0.13	1,135
5%	24	2,024	0.13	1,265
5%	25	2,183	0.13	2,802
5%	26	2,348	0.13	1,524
5%	27	2,519	0.13	2,794
5%	28	2,696	0.13	7,421
5%	29	2,879	0.13	2,051
5%	30	3,195	0.13	3,112
5%	31	3,394	0.13	1,644
5%	32	3,599	0.13	2,687
5%	33	3,810	0.13	3,892
5%	34	4,027	0.13	3,162
5%	35	4,250	0.13	3,422
5%	36	4,479	0.13	9,624
5%	37	4,714	0.13	5,199
5%	38	4,955	0.13	4,188
5%	39	5,202	0.13	16,519
5%	40	5,624	0.13	8,209
5%	41	5,887	0.13	5,167
5%	42	6,156	0.13	9,280
5%	43	6,431	0.13	9,845
5%	44	6,712	0.13	52,209
5%	45	6,999	0.13	20,152
5%	46	7,292	0.13	14,047

5%	47	7,591	0.13	32,373
5%	48	7,896	0.13	7,898
5%	49	8,207	0.13	25,169
5%	50	8,735	0.13	60,369
5%	51	9,062	0.13	12,548
5%	52	9,395	0.13	19,894
5%	53	9,734	0.13	20,900
5%	54	10,079	0.13	80,883
5%	55	10,430	0.13	19,430
5%	56	10,787	0.13	20,262
5%	57	11,150	0.13	17,105
5%	58	11,519	0.13	53,654
5%	59	11,894	0.13	28,133
5%	60	12,528	0.13	124,379
5%	61	12,919	0.13	232,533
5%	62	13,316	0.13	421,696
5%	63	13,719	0.13	146,818
5%	64	14,128	0.13	117,965
5%	65	14,543	0.13	431,594
5%	66	14,964	0.13	160,037
5%	67	15,391	0.13	158,905
5%	68	15,824	0.13	294,006
5%	69	16,263	0.13	270,713
5%	70	17,003	0.13	2,046,015
5%	71	17,458	0.13	155,812
5%	72	17,919	0.13	440,564
5%	73	18,386	0.13	1,103,835
5%	74	18,859	0.13	155,497
5%	75	19,338	0.13	1,645,749
5%	76	19,823	0.13	1,485,769
5%	77	20,314	0.13	501,377

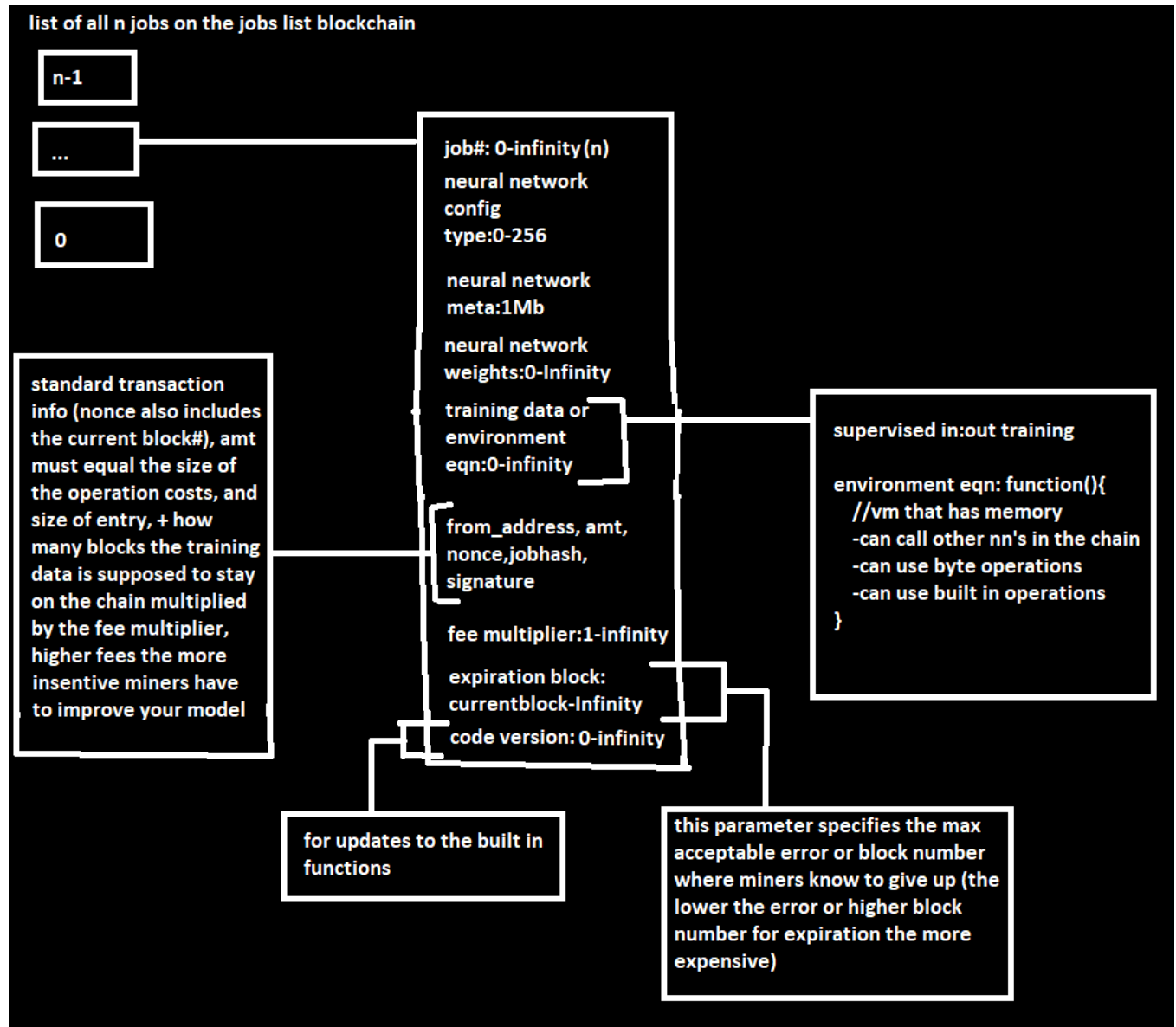
5%	78	20,811	0.13	867,643
5%	79	21,314	0.13	493,950
5%	80	22,160	0.13	1,289,056
5%	81	22,679	0.13	1,148,156
5%	82	23,204	0.13	1,941,372
5%	83	23,735	0.13	4,366,564
5%	84	24,272	0.13	1,475,820
5%	85	24,815	0.13	4,827,143
5%	86	25,364	0.13	5,784,586
5%	87	25,919	0.13	3,065, 207
5%	88	26,480	0.13	3,747,722
5%	89	27,047	0.13	19,382,402
5%	90	27,999	0.13	5,483,653
5%	91	28,582	0.13	7,156,542
5%	92	29,171	0.13	598,870
5%	92	29,766	0.13	?

Sections to add to wp:

1. Storing block solutions on the blockchains for a block of difficulty 90 would take about 120kB per block, and because solutions need to be run over a training data set to check the solution it makes miners more susceptible to DDOS attacks.
2. Training data sets can be large – how can miners enforce the maintenance of massive training data to the network. Training data would have to pay rent to be kept alive on the block chain, and part of a requirement to submit a new block means you must include information from training data that is still paying rent to be kept on the blockchain while miners try to solve the problem.
3. The cost for uploading custom problems is determined by the current difficulty of the network, as well as the fact that miners can choose to attempt your problem or not
4. How to make the system constantly adaptable? How are new functions incorporated into the problem generation? What are the domain of variables that can be changed in procedural problem generation:
 - a. Activation functions
 - i. Functions are kept on the chain and all referenced in an index
 - ii. Index also keeps track of the frequency the operation is used by user submitted problems *this can be manipulated to potentially stall the network by making a complex activation function need to define the rules of this function at the byte operation level
 - iii. and based on the percentage they are used they are incorporated into the

- b. Number of layers
- c. Connectivity of layers
- d. Base equation the nn is trying to approximate

Data Model:



Miner Steps:

miner lifecycle

