Use Cases

For the Team Tensor Capstone Project, we need a Transport-Layer protocol that satisfies the following requirements:

1. The Data Source can assign a data pipeline to any worker
2. The Data Source can assign a model to any worker
    a. This also includes specifying that a given model belongs to a data pipeline
    b. We also assume that a data pipeline was assigned to a given worker before a model belonging to the pipeline was assigned to the board
3. The Data Source can upload a model to the worker board
    a. To avoid having to implement model initialization in hardware, the initial weights of the model will be specified by the Data Source machine
    b. This also includes specifying metrics and objectives
4. The Data Source can probe a worker's model manager to determine if there is space in the manager's batch queue
5. The Data Source can send a batch of input data points to a given model manager on a given worker
6. The Data Source can retrieve metric information from a given worker
7. The Data Source can retrieve the model weights for a specific model

The protocol is to be implemented as an overlay protocol for UDP.

When thinking of training a single model, there are four phases:

1. Define model specifications
2. Train the model
3. Evaluate the model
4. Retrieve the weights

In addition, we need a way for the Data Source and Worker machines to make initial contact with each other given no prior information about the network. This must be done before any of the above phases can be entered, so we call it the "Hello" phase.

Therefore, interactions between the worker and the source can be classified into any of five categories:

1. Hello
2. Model definition
3. Model training
4. Model evaluation
5. Model retrieval

Each worker machine is agnostic to the interactions between the Data Source and any other worker, and therefore it is agnostic to the states of the other workers. Each model manager is

also agnostic to the states of any other model managers on the same worker, including the model managers belonging to the same data pipeline.

Each interaction exists as a series of IP packets being transmitted between the worker and the Data Source machine. The following are all defined interactions:

1. Worker asks if a Data Source is listening on the bus
2. Data Source assigns a data pipeline to a worker
3. Data Source assigns a model to a worker
4. Data Source probes a worker to learn the number of model managers remaining
5. Data Source probes a worker's model manager to learn if there is enough space in the batch queue to fit another batch
    a. Occurs in Train and Evaluation phases
6. Data Source sends a batch to a worker
    a. Occurs in Train and Evaluation phases
7. Data Source requests a metric from a worker
    a. Occurs in Train and Evaluation phases
8. Data Source asks a worker for a model's weights

Each interaction is an atomic sequence of packets with specific senders and destinations sent in a specific order. In general, a packet is a stream of bytes where the first byte is an opcode header that identifies the purpose of the packet. A packet can be of any length, although this specification will specify an exact length for most packets. Packets that can actually be of arbitrary length will have fields that deterministically indicate how the packet must be interpreted.

Packet Specifics

The packet opcode headers are as follows:

- HELLO
    - Request a response from active Data Source(s) on the network
    - 0x01
- ACK
    - Acknowledge receipt, request is approved
    - 0x02
- N_ACK
    - Acknowledge receipt, request is denied
    - 0x03
- ASN_DP
    - Assign a data pipeline
    - 0x04

- ASN_MD
    - Assign a model
    - 0x05
- M_FULL
    - Check if there are any available model managers on a worker
    - 0x06
- B_FULL
    - Check if there is enough space in a model's batch queue for another batch
    - 0x07
- BATCH
    - Send a batch to a worker
    - 0x08
- GET_MT
    - Retrieve a metric from a worker
    - 0x09
- GET_MD
    - Retrieve a model descriptor from a worker
    - 0x0a
- BATCH
    - Upload a batch of inputs to the worker
    - 0x0b

Every packet header is exactly one byte.

Each of the interactions is defined as follows:

Data Source asks if a Worker is listening on the bus

1. Data Source sends

1 byte
HELLO

2. Worker sends

1 byte
ACK

If the Data Source does not receive a response within a certain amount of time (e.g. 50 milliseconds), then the Data Source rebroadcasts the message, and repeats ad infinitum.

Data Source assigns a data pipeline to a worker

1. Data Source sends

1 byte	2 bytes
ASN_DP	<data pipeline id>

2. Worker sends

1 byte	2 bytes
ACK	<data pipeline id>

Data Source assigns a model to a worker

1. Data Source sends

1 byte	2 bytes	2 bytes	4 bytes	4 bytes	Arbitrary length	1 byte	Arbitrary length
ASN_MD	<data pipeline id>	<model id>	<size of model descriptor>	<Number of layers>	<model descriptor>	<number of metrics>	<metrics>

Model Descriptor includes:

-

2. Worker calculates the number of models assigned to the given data pipeline and sends
    a. Number of models response is more of a debugging tool to make sure the Data source machine and board are on the same page with # of models

1 byte	2 bytes

ACK	<number of models>

Data Source probes a worker to learn the number of model managers remaining

1. Data Source sends

1 byte
M_FULL

2. Worker sends

1 byte	2 bytes
ACK	<number of model managers available>

Data Source probes a worker's data pipeline router to learn if there is enough space in the batch queue to fit another batch

1. Data Source sends

1 byte
B_FULL

2. Worker sends

1 byte
ACK

if the model manager has space and otherwise sends

1 byte
NACK

Data Source sends a batch to a worker

Assume that each sample is the same size, where each sample is a single tensor, serialized according to the protocol at the bottom of this document.

1 byte	2 bytes	Arbitrary length	...	Arbitrary length
BATCH	<number of inputs>	<sample 1>	...	<sample n>

Data Source requests a metric from a worker

1. Data Source sends

1 byte	1 byte
GET_MT	Metric ID(accuracy or loss)

2. Worker sends

1 byte	Floating point size
ACK	Value of metric

Data Source asks a worker for a model's weights

1. Data Source sends

1 byte	2 byte
GET_MD	<model id>

2. Worker sends

1 byte	Arbitrary length
ACK	<model descriptor>

Representation of Models and Tensors

It is also important to set in stone how we will serialize tensors and models, given that these data are of arbitrary size.

Tensors will be stored in column-major order. For example, given a point T[i][k] in memory, the point to the right is the point T[i + 1][k] if i + 1 is within the size of the first dimension of tensor T, and T[i][k + 1] otherwise. Storing the tensors in a column-major format will allow for easy implementation of matrix-vector multiplication in SystemVerilog, and adhering to this at higher levels of the architecture will reduce the need for matrix transposition at any level of the memory hierarchy.

Filters for convolutional layers will have the shape (output channels, input channels, height, width). Weight matrices for linear layers will have the shape (height, width).

Tensors can have an arbitrary number of dimensions and each dimension can be of arbitrary size. Therefore, a tensor will be serialized as follows:

1 byte	2 bytes	...	2 bytes	Arbitrary
<number of dimensions>	<size of dimension 1>	...	<size of dimension n>	<Tensor descriptor, as described above>

For example:

```
Matrix of integers:

10   16
4    8
6    3
```

```
Serializes to:

0x02  0x0003  0x0002 0x0000000a 0x00000004 0x00000006 0x00000010
0x00000008 0x00000003

The integer image (with three color channels):

Channel 1
10   16
4    8
6    3
Channel 2:
11   17
5    9
7    4
Channel 3:
12   18
6    10
8    5
Channel 4:
13   19
7    11
9    6

Serializes to:

0x03 0x0004 0x0003 0x0002 0x0000000a 0x0000000b 0x0000000c 0x0000000d
0x00000004 0x00000005 0x00000006 0x00000007 0x00000006 0x00000007
0x00000008 0x00000009 0x00000010 0x00000011 0x00000012 0x00000013
0x00000008 0x00000009 0x0000000a 0x0000000b 0x00000003 0x00000004
0x00000005 0x00000006
```

Models will be stored similarly. The first layer in the model appears as the leftmost model in the serial bitstream.

1 byte	1 bytes	arbitrary	...	1 byte	1 byte	...
<number of layers>	<layer 1 code>	<layer 1 weights>	...	<number of metrics, first is	<metric codes>	...

				objective>		

The layer code will represent which type of layer this is, and will thus influence how to interpret the following bytes. Supported layers are:

Linear	0x01
2D Convolution	0x02
ReLU	0x03
MaxPool	0x04
Flatten	0x05
Softmax	0x06

In addition, the model descriptor will specify all of the metrics that will be used to evaluate the model. The first one is the objective (loss) function, so it is required that a model descriptor specify a metric.

Also, when serializing a convolutional layer, there must be an extra 4 words (16 bytes) that describe the pad, stride, width, and height of the image to be output by the convolutional layer. These 4 words will be serialized immediately after the layer opcode and before the weight tensor.