Bittensor

Release 1

Oct 08, 2020

Getting Started

Welcome to the official documentation of Bittensor, the peer-to-peer machine intelligence framework.

The dominant tools used to guide machine intelligence are typically benchmarks which rank systems on a set of predefined tasks. However, the narrow low resolution definition of intelligence provided by these benchmarks make them inefficient guides for the field. Bittensor is an expanded framework that measures knowledge production from within a network of intelligence systems.

CHAPTER 1

Installation

1. Install or make sure you have installed Python3.
2. Install or make sure you have installed pip3.
3. If you do not have it installed already, install virtualenv.
4. If you wish to run Bittensor in dockerized containers, install Docker Desktop.
5. Clone the Bittensor repository locally:

```
git clone git@github.com:opentensor/bittensor.git
```

6. If you wish to simply run a dockerized Bittensor instance, you may do so by running:

```
./start_bittensor.sh
```

This will set up and bootstrap Bittensor in a local Docker container. If the process succeeds, you should see a "Bittensor" ASCII banner as the model is bootstrapped and run on docker. The default model that runs in this case is the "MNIST" dataset model. This is an example model that is set up under `examples/mnist`. You can specify other models by using the `-n, --neuron` flag.

7. Create a new virtual environment and activate it:

```
python3 -m venv env
source env/bin/activate
```

8. Install required packages:

```
pip3 install -r requirements.txt
```

9. Install protobufs

```
pip3 install -e .
```

10. You can verify your installation has succeeded by going to one of the examples and running its `main.py` folder.

```
cd examples/mnist
python3 main.py
```

Similarly to step 6, this will run the example with all the default flags. If everything installed as it should, you should see an ASCII “Bittensor” printed before the model starts training.

CHAPTER 2

Run Bittensor via Docker

Since Docker containers encapsulate everything an application needs to run (and only those things), any host with the Docker runtime installed (be it a developer's laptop or a public cloud instance) can run a Docker container. This is very useful when deploying your model to a cloud GPU or some remote peer if you do not wish to handle the hassle of setting up an environment locally. Hence, once a model has been trained and is ready to be deployed onto the Bittensor network, you may wish to use Docker to deploy it somewhere instead of a native Python call. For this purpose, Bittensor contains a bash script (`start_bittensor.sh`) that automatically pulls the latest image from Docker Hub and constructs a docker container that is able to run a given model. The parameters of `start_bittensor.sh` are summarized as follows:

Flag	Description
`-n, --neuron`	Which model (or "neuron") to run in `examples/`. E.g. `mnist`, `cifar`, `bert`, etc.
`-l, --logdir`	Logging directory
`-p, --port`	Bind side port for accepting requests.
`-c, --chain_endpoint`	Bittensor chain endpoint.
`-a, --axon_port`	Axon terminal bind port.
`-m, --metagraph_port`	Metagraph bind port.
`-s, --metagraph_size`	Metagraph cache size.
`-b, --bootstrap`	Metagraph boot peer.
`-k, --neuron_key`	Neuron Key.
`-r, --remote_ip`	Remote serving IP.
`-mp, --model_path`	Path to a saved version of the model to resume training.

To start a dockerized version of a model, simply run the following:

```
./start_bittensor.sh
```

This will utilize the default values for all of the above command line arguments and run the network under `examples/mnist` on `localhost`.

CHAPTER 3

Run Bittensor via Python

First, ensure that you have `Python 3.5` or above to run Bittensor. Once you have done so, you can move to the `examples` directory and pick a model to run. You can run the model directly by calling:

```
python main.py
```

This will run a single instance of the model that will not speak to any peers with default parameters. The parameters for running a Bittensor model are as follows:

Flag	Description
`--chain_endpoint`	Bittensor chain endpoint.
`--axon_port`	Axon terminal bind port.
`--metagraph_port`	Metagraph bind port.
`--metagraph_size`	Metagraph cache size.
`--bootstrap`	Metagraph boot peer.
`--neuron_key`	Neuron Key.
`--remote_ip`	Remote serving IP.
`--model_path`	Path to a saved version of the model to resume training.

To run a model on a given port number (e.g. 8120) that peers can connect to, you can run it as follows:

```
python main.py --metagraph_port 8120
```

Once this model is running, you can run another model that bootstraps itself to port 8120 as follows:

```
python main.py --bootstrap '0.0.0.0:8120'
```

This will run another instance of the model that will communicate with the first model and they will begin to share knowledge with each other.

CHAPTER 4

Running Multiple Bittensor Instances

The advantage of Bittensor is that it allows for *knowledge sharing* between models training together. This not only allows for more efficient training, but also increases the possibilities for us to develop resilient generalist models, as opposed to narrow specialists.

Let's dive into how we can test running two models at the same time that can share knowledge with each other as they train. Bittensor utilizes gRPC communication to send PIL data, tensors, and gradients across the wire from one model to another model (also called a "peer"). We will first show how we can achieve this using Docker containers, and then show the same process using native Python commands.

Each Bittensor instance (native Python or Dockerized) requires some specific ports to be opened up for it to be able to communicate with other instances. These are:

1. **Axon Port** : This port is where communications are received. If it is not opened, then your bittensor instance cannot receive gRPC communications from its peers. This is automatically exposed on a random port number on the Docker container via the `start_bittensor.sh` script if running on Docker, and via Python as well if running on native Python. Hence, there is typically no need to set it yourself unless you are going after some rather specific architectural setup for your network.

2. **Metagraph Port** : This port is largely for testing purposes, and is responsible for specifying which metagraph port to connect to. In a nutshell, the metagraph is the connecting fabric between all the nodes/models and acts as a "simulated" chain when a blockchain is not present. When testing locally, the first instance you run needs to specify its metagraph port.

3. **Bootstrap port** : When testing, each node will need to bootstrap itself to a peer so they can begin the communication process. When testing locally on Docker, the IP of this peer automatically resolves to `host.docker.internal`. When testing via native Python, the IP is specified as `0.0.0.0`. When testing locally, this port needs to be set to be the same as the metagraph port that the first instance you launched is using.

4.1 Via Dockerized Containers

To run multiple instances via Docker containers, we must expose the Axon, Metagraph, and Bootstrap ports. We must also first create an instance that binds itself to the Metagraph via a given port, and the second instance must then bootstrap itself to the first instance to begin communication.

Let's start by running the first instance and binding it to metagraph port 8120.

```
./start_bittensor.sh -m 8120
```

This starts a Dockerized instance of Bittensor bound to metagraph port 8120 without a bootstrapped peer.

The training output for each iteration will look like the following:

```
2020-10-06 14:21:59.100 | INFO     | __main__:train:291 - Train Epoch: 0 [0/60000 (0
↪%)]     Local Loss: 2.306969    Target Loss: 2.306806   Distillation Loss: 0.027752␣
↪    nP|nS: 0|1
```

Note that the very last values of this line: nP|nS: 0|1. This indicates the number of peers (nP) and the number of Synapses (nS), at the moment there are no peers and one synapse that belongs to this instance that we've just launched.

Now let's launch another instance that will communicate with the initial instance we just launched. The appropriate bootstrap IP is automatically determined when running locally, so we just need to specify the bootstrap_port. Open another terminal and run the following call:

```
./start_bittensor.sh -b 8120
```

Now if we take a look at the training output, note that nP and nS changed to nP|nS: 1|2 as there is one peer and two synapses firing now.

```
2020-10-06 14:22:08.616 | INFO     | __main__:train:291 - Train Epoch: 0 [0/60000 (0
↪%)]     Local Loss: 2.299487    Target Loss: 2.299433   Distillation Loss: 0.038577␣
↪    nP|nS: 1|2
```

This indicates that our new instance has bootstrapped successfully to the first instance that we launched, and is communicating with it appropriately.

4.2 Via Native Python

The process to run multiple instances via native Python is very similar to the Dockerized example, the only difference being the syntax of the command line arguments as we are sending them directly to Python in this case. As in the Docker container example, we first create an instance that binds itself to the Metagraph via a given port, and the second instance must then bootstrap itself to the first instance to begin communication.

Let's start by running the first instance and binding it to metagraph port 8120.

```
python3 examples/mnist/main.py --metagraph_port=8120
```

Now let's launch another instance that will communicate with the initial instance we just launched. The difference from Docker, however, is that the appropriate bootstrap IP is **not** determined automatically now, so we must specify it along with the bootstrap port. Open another terminal and run the following call:

```
python3 ./examples/mnist/main.py --bootstrap='0.0.0.0:8120'
```

The output from the training loops both instances will be the same as the dockerized versions:

The training output for each iteration of the first instance will look like the following:

```
2020-10-06 14:21:59.100 | INFO     | __main__:train:291 - Train Epoch: 0 [0/60000 (0
↪%)]     Local Loss: 2.306969    Target Loss: 2.306806   Distillation Loss: 0.027752␣
↪    nP|nS: 0|1
```

whereas the training output for each iteration of the second (boostrapped) instance will look like:

```
2020-10-06 14:22:08.616 | INFO     | __main__:train:291 - Train Epoch: 0 [0/60000 (0
↪%)]     Local Loss: 2.299487    Target Loss: 2.299433   Distillation Loss: 0.038577␣
↪     nP|nS: 1|2
```

CHAPTER 5

gRPC Protocol

The BitTensor gRPC protocol definition can be found here. The following define the communications protocol and message types that travel between nodes/instances of Bittensor.

5.1 `Bittensor` service.

- Service definition for tensor processing servers.
- RPC Methods:

Method	Description	Returns
Forward (`TensorMessage`)	Forward tensor request.	`TensorMessage`
Backward (`TensorMessage`)	Backward tensor request (i.e. pass gradients back)	`TensorMessage`

5.2 `Metagraph` service.

- Service definition for a cache for sharing bittensor synapses.
- This will be replaced by a Blockchain in the future, and **is for testing purposes only.**

Method	Description	Returns
Gossip (`GossipBatch`)	Use gossip protocol to find other peers on the network.	`GossipBatch`

5.3 `GossipBatch` message.

- Batch of neuron service definitions.
- Gossip message sent across the wire to bootstrap to peers.

GossipBatch Payload	Description
`STRING` version [required]	Identifies protocol version for backward compatibility.
`REPEATED STRING` peers [required]	List of metagraph peer addresses.
`SYNAPSE` synapses	Synapse endpoint definitions.

5.4 `DataType` enum.

- Used for serialization/deserialization of messages as they travel through the wire.
- Data types can be `UNKNOWN`, `FLOAT32`, `INT32`, `INT64`, or `UTF8`.
- Described here.

5.5 `Modality` enum.

- Modality of the message (TEXT, IMAGE, TENSOR).
- Described here.

5.6 `TensorMessage` message.

- The primary protobuf message object passed between tensor processing servers.
- **Contains a payload of 1 or more serialized tensors and their definitions.**
  - **version** – Indentifies protocol version for backward compatibility.
  - **neuron_key** – Public key of the caller.
- Also contains information to identity and verify the sender.

TensorMessage Payload	Description
`STRING` version [required]	Identifies protocol version for backward compatibility.
`STRING` neuron_key [required]	Public key of the calling server. This is used to identify the calling server (i.e. neuron) to the synapse of the receiving server (i.e. the receiving synapse).
`STRING` synapse_key [required]	Public key of the receiving synapse. This is used to identify which synapse to query behind the endpoint, as an endpoint may have multiple synapses.
`INT64` nounce [optional]	Incrementing nounce to identify message ordering. Used ensure with signature to protect against spoofing attacks.
`BYTES` signature [optional]	Digital signature linking the nounce, neuron_key and synapse_key. This prevents spoofing attacks where an adversary sends messages pretending to be other peers.
`TENSOR` tensors [required]	1 or more tensors passed on the wire. This is the whole point of having a TensorMessage, and thus is required.

5.7 `Tensor` message.

- A serialized tensor object created using the serializer class.
- Essentially used to describe a tensor being passed on the wire.

Tensor Payload	Description
`STRING` version [required]	Identifies protocol version for backward compatibility.
`BYTES` buffer [required]	Serialized raw tensor content. Since this is serialized, this representation can be used for all tensor types. The purpose of this serialization is to reduce overhead during RPC calls by avoiding the serialization of many repeated small items.
`repeated INT64` shape [required]	Tensor shape.
`DataType` dtype [required]	The tensor datatype. Used for serialization and deserialization.
`Modality` modality [required]	Modality of the message (TEXT, IMAGE, TENSOR)
`bool` requires_grad [optional]	Whether or not this tensor require a gradient.

5.8 `Synapse` message.

- "Synapse" or "Expert" endpoint definition.
- This fully describes a tensor processing service for Bittensor (as well as hivemind).

Synapse Payload	Description
STRING version [required]	Identifies protocol version for backward compatibility.
STRING neuron_key [required]	Public key of the calling neuron. This is used to identify the calling server (i.e. neuron) to the synapse of the receiving server (i.e. the receiving synapse). Links this synapse definition to the containing neuron-account.
STRING synapse_key [required]	Public key of this synapse. This is used to identify which synapse to query behind the endpoint, as an endpoint may have multiple synapses.
STRING address [required]	Synapse ip address.
STRING port [required]	Synapse endpoint listening port.
STRING block_hash [required]	Hash of latest chain block service definition creation.
STRING nounce [optional]	Incrementing nounce.
STRING proof_of_work [optional]	Work hash which ensures service definition creation was computationally expensive. Protects the network cache from flooding attacks.
STRING signature [optional]	Public_key signature for this synapse definition. Ensures that the connected neuron_key signed this proto.

CHAPTER 6

Bittensor Architecture

The inspiration for Bittensor came from brain cells (called neurons). In the same way that a network of neurons consists of multiple neurons communicating with each other continuously, so does the Bittensor network consist of multiple machine learning models (also called neurons) communicating with each other.

Therefore to help in understanding the Bittensor network architecture, we use biological neurons as inspiration and an analogous example.

6.1 Biological Inspiration

The Figure below describes a biological neuron that contains three important parts:

1. **Dendrite**: responsible for receiving information from other neurons.
2. **Axon Terminal**: responsible for sending information to other neurons.
3. **Soma**: Contains the nucleus and other structures common to living cells.

These three parts together enable a neuron to exchange messages with each other and form a massive interconnected network that – on the grand scale – form a brain. Note that this is a highly simplified explanation, and biological neurons are significantly more complex.

6.2 The Bittensor Network

Similarly to its biological counterpart, the Bittensor neuron also exchanges messages with other neurons in the form of *machine knowledge*. A group of these neurons can connect together to form a network of interconnected models that are able to learn from each other.

Bittensor Neuron examples can be found under `examples`. Presently, there are 3 examples:

- MNIST : The most basic neuron example. A simple feed-forward network that takes 32x32 images as input.
- CIFAR : A more complicated Dual Path Neural Network (As described by Chen et al.) that takes as input the CIFAR-10 dataset.

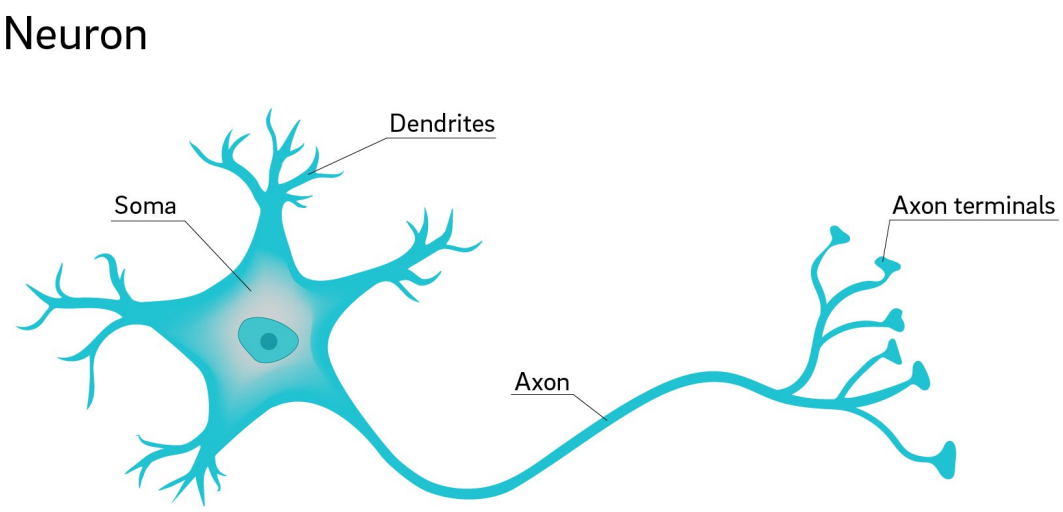

Fig. 1: Biological neuron cell. Credit: David Baillot/ UC San Diego.

- BERT : A complete implementation of Huggingface's bert model.

For the rest of this document, we will refer to **MNIST** during our examples as it is the most straightforward network, and hence easier to understand the architecture of Bittensor through it.

Assume we have two neurons as in the figure below, where each neuron contains 3 main components:

1. **Axon Terminal**: Responsible for deploying a synapse and receiving information coming from a remote synapse.
2. **Dendrite**: Responsible for sending information to a remote synapse.
3. **Neuron**: Effectively the "soma" of a bittensor node. Contains the local model as well as the training and testing logic.

Let's also assume that as per the *getting started* guide, we started Neuron 1 first.

Let's go through a breakdown of what happens when we start a Neuron. You can follow along in the MNIST code.

1. Neuron loads its configuration (set by the command line argumnets) and dataset, then sets up the hyper parameters such as the learning rate, batch size, etc.

```
# Additional training params.
batch_size_train = 64
batch_size_test = 64
learning_rate = 0.01
momentum = 0.9
...
# Load (Train, Test) datasets into memory.
train_data = torchvision.datasets.MNIST(root=model_toolbox.data_path,␣
↪train=True, download=True, transform=transforms.ToTensor())
trainloader = torch.utils.data.DataLoader(train_data, batch_size = batch_
↪size_train, shuffle=True, num_workers=2)
```

(continues on next page)

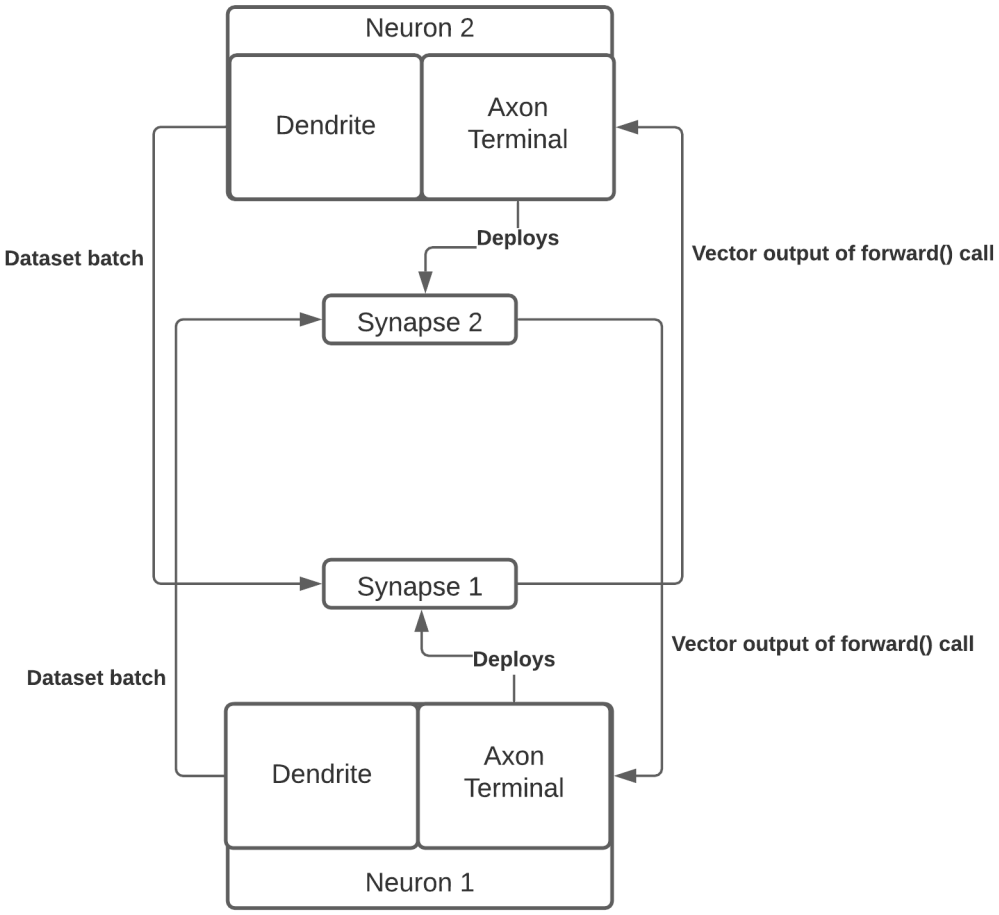

Fig. 2: Communication between two Bittensor neurons.

(continued from previous page)

```
test_data = torchvision.datasets.MNIST(root=model_toolbox.data_path,␣
↪train=False, download=True, transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(test_data, batch_size = batch_
↪size_test, shuffle=False, num_workers=2)
```

2. Neuron builds the local synapse (this is the model to be trained on the network).

```
# Build local synapse to serve on the network.
model = MnistSynapse() # Synapses take a config object.
model.to( device ) # Send model to device (GPU or CPU)
```

   where `MnistSynapse` is a class that extends the `Synapse` class. In true Pytorch fasion, it contains an `__init__` and a `forward()` call.

3. The Neuron will then build and start the `Metagraph` object. This object is responsible for connecting to the Blockchain and finding other neurons on the network.

```
metagraph = bittensor.Metagraph( config )
metagraph.subscribe( model ) # Adds the synapse to the metagraph.
metagraph.start() # Starts the metagraph gossip threads.
```

4. The Axon server is built next. This server allows other neurons (Neuron 2 in this example) to make queries to Neuron 1 through the Dendrite. It also deploys the synapse model we set up in step 2.

```
axon = bittensor.Axon( config )
axon.serve( copy.deepcopy(model) )
axon.start() # Starts the server background threads. Must be paired with␣
↪axon.stop().
```

5. Neuron builds the Dendrite object next. The Dendrite is responsible for sending dataset batches across the network to remote synapses.

```
dendrite = bittensor.Dendrite( config ).to(device)
```

6. Finally, the Neuron builds the router. The router is responsible for learning **which** synapse to call.

```
router = bittensor.Router(x_dim = 1024, key_dim = 100, topk = 10)
```

7. We can set up the optimizer the same way we normally do with any other Pytorch model. The important piece here is that we are optimizing both the model parameters **and** the router's parameters, as we are dealing with two models here. The Synapse model that we are training, and the router's model that learns which synapse model to tell the Dendrite to send a dataset batch to.

```
# Build the optimizer.
params = list(router.parameters()) + list(model.parameters())
optimizer = optim.SGD(params, lr=learning_rate, momentum=momentum)
```

8. If we have previously saved a Bittensor model and wish to continue training it, we can load it back up using the `model_toolbox`.

```
# Load previously trained model if it exists
if config._hparams.load_model is not None:
    model, optimizer, epoch, best_test_loss = model_toolbox.load_
↪model(model, config._hparams.load_model, optimizer)
```

9. The training loop proceeds as it would with static training models that we know and love in Pytorch. However there is one difference now that they are communicating with each other: we need to actually tell them to talk to each other during training. Hence, during the training loop we invoke the following lines:

```
# Encode inputs for the router context.
context = model.forward_image(images).to(device)
```

Note that by `context` here we mean the input coming from the local network, which is a vector output of its forward pass. This same context is used as input for the remote networks.

This invokes the model for one forward pass of the image inputs through the `MnistSynapse` model we defined. We then query the network of peers and send them the vector output of this forward pass and the current batch of examples that we are training on.

```
# Query the remote network of peers
synapses = metagraph.get_synapses( 1000 ) # Returns a list of synapses on␣
↪the network (max 1000).
requests, scores = router.route( synapses, context, images ) # routes␣
↪inputs to network.
responses = dendrite.forward_image( synapses, requests ) # Makes network␣
↪calls.
network = router.join( responses ) # Joins responses based on scores..
```

Let's unpack this line by line:

- `metagraph.get_synapses()` will simply query the network, and return a list of synapses that are presently on the network that we can query. Recall that a remote synapse is a model running on a remote neuron.
- `router.route()` will utilize the router model to find which synapses to query that will return the best responses. It will return `requests`: a list of input minibatches to be sent to the remote synapses, and `scores`: a list of scores of the performance of the remote synapses.
- `dendrite.forward_image()` will forward the minibatches to the remote synapses, and return their responses – a vector output of their `forward` call.
- `router.join()` will join the responses of all the synapses together.

NOTE: If we are running only one instance of Bittensor with no peers, then this call would simply recursively send the batch and vector output of the forward pass to the instance itself, effectively learning locally as if it's a local learning model.

10. Now that we have the responses from all the remote synapses, we can call `forward()` on the local model to calculate the following:
    - `loss`: Total loss acumulation to be used by `loss.backward()`.
    - `local_output`: Output encoding of image inputs produced by using the local distillation model as context rather than the network.
    - `local_target`: MNIST Target predictions using student model as context.
    - `local_target_loss`: MNIST Classification loss computed using the local_output, student model and passed labels.
    - `network_target`: MNIST Target predictions using the network as context.
    - `network_output`: Output encoding of inputs produced by using the network inputs as context to the model rather than the student.
    - `network_target_loss`: MNIST Classification loss computed using the local_output and passed labels.

- `distillation_loss`: Distillation loss produced by the student with respect to the network context.

Remember that by `context` here we mean the input coming from the remote network of neurons, which is a vector output of their own forward passes. The student distillation model learns to emulate this context from the network as well to increase accuracy and reduce loss.

11. Finally, let's set the weights of the remote synapses. We first get the weights for list of Synapse endpoints, normalize them by the scores we calculated for the remote synapses, then we set them back. This process is what allows a Bittensor neuron pick the best remote synapses (or peers) upon the next training iteration.

```
weights = metagraph.getweights(synapses).to(device)
weights = (0.99) * weights + 0.01 * torch.mean(scores, dim=0)
metagraph.setweights(synapses, weights)
```

CHAPTER 7

Bittensor Component Breakdown

Bittensor consists of several components that allow neurons to communicate with peers (other neurons) and send/receive machine knowledge. This section goes through each component and describes its role in the neuron, its inputs , its outputs, and its context within the big picture of the Bittensor network.

These components can all be found in bittensor/bittensor.

7.1 Axon class

Analogous to the Axon terminal in a biological neuron cell, the axon class is responsible for deploying a Synapse instance and receiving machine knowledge from other synapses in the network (also called remote synapses, we will be using these terms interchangeably).

The main job of the Axon class is to process `forward` and `backward` requests through a set of local synapses. This requests come from remote neurons.

7.1.1 `__init__ (self, config: bittensor.Config)`

- Initializes the configuration according to arguments passed to `main.py` in the Neuron.
- Initializes gRPC server objects using the `config.axon_port` passed to it.
- Sets up local synapses and the serving Axon thread.

7.1.2 `__del__ (self)`

- Stops the entire Axon terminal server and deletes local synapses.

7.1.3 `_stop (self)`

- Stop the Axon terminal and the gRPC server.

7.1.4 _serve (self)

- Start the gRPC server.

7.1.5 serve (self)

- Add a synapse to the set of synapses presently being served by this Axon terminal.
- Creates a new bittensor_pb2.Synapse proto and adds it to the list of local synapses.

7.1.6 Forward (self, request, context)

The `Forward` call takes the *TensorMessage* request and looks up the local synapse that this request is targetting. In this context, the local synapse is a synapse belonging to this neuron. Once it has found the synapse, it deserializes the Tensor in the *TensorMessage* and makes a `forward` call on the target synapse. It then serializes the local synapse's response (a vector of the output of its last layer) and returns it.

Inputs:

- *TensorMessage request* coming from a remote synapse.
- `congrpc.ServicerContext context` : coming from a remote neuron.

Outputs:

- Serialized response of the local synapse.

7.2 Dendrite class

Also analogous to the Dendrite in a biological neuron cell, the Dendrite class is responsible for sending dataset batches across the network to remote synapses.

7.2.1 forward_text, forward_image, and forward_tensor

As the function name implies, each of these functions is responsible for forwarding a different modality across the gRPC network to peer neurons.

Each function takes as input a list of peer synapses and a list of torch tensors that need to be sent to the peer. The list of peer synapses can be retrieved with a call to `metagraph.get_synapses()` as described in *Bittensor Architecture*.

```
def forward_text(self, synapses: List[bittensor_pb2.Synapse], x: List[ List[str] ]) ->
↪ List[torch.Tensor]:
    """ forward tensor processes """
    return self.forward(synapses, x, bittensor_pb2.Modality.TEXT)

def forward_image(self, synapses: List[bittensor_pb2.Synapse], x: List[ torch.Tensor␣
↪]) -> List[torch.Tensor]:
    """ forward tensor processes """
    return self.forward(synapses, x, bittensor_pb2.Modality.IMAGE)

def forward_tensor(self, synapses: List[bittensor_pb2.Synapse], x: List[ torch.Tensor␣
↪]) -> List[torch.Tensor]:
    """ forward tensor processes """
    return self.forward(synapses, x, bittensor_pb2.Modality.TENSOR)
```

7.2.2 forward (self, synapses, x, mode)

The main functionality behind the Dendrite component of Bittensor, the `forward()` call forwards mini-batches of the dataset by invoking `_RemoteModuleCall` out to the target remote synapses and returns their results.

Inputs:

- `List[bittensor_pb2.Synapse] synapses` : A list of remote synapses running in remote neurons.
- `List[ object ] x` : A list of objects to be sent through gRPC to the target remote synapses.
- `bittensor_pb2.Modality mode` : Modality of the data being sent (i.e. Text, Image, etc.)

Outputs:

- The response of each remote synapse. Given that each minibatch runs through the remote synapse's local model, the output back is a vector output of that remote synapse's `forward()` call.

7.2.3 RemoteSynapse class

This class bundles a grpc connection to a remote neuron as a standard auto-grad torch.nn.Module. Making it possible to send data to that neuron and even calculate its partial derivative on a backwards pass.

__init__ (self, synapses, config)

This function sets up the synapse's remote address and a gRPC channel for communication.

Inputs:

- `bittensor_pb2.Synapse synapse`: The target remote synapse to which we are bundling a gRPC connection.
- `bittensor.Config config` : Run configuration of the local neuron.

Outputs:

- `None`

forward (self, object, mode)

Makes a `_RemoteModuleCall` to send the batch of inputs over to the remote synapse.

Inputs:

- **`Object inputs`** [The mini-batch of data to be sent to the remote synapse. Note that it is of type `object` as] it can be of type string or image data. More data types will be integrated in later versions.
- `mode` : The modality of the data being sent.

Outputs

- `torch.Tensor outputs` : A Tensor of outputs from the remote synapses.

7.2.4 _RemoteModuleCall class

This class is responsible for invoking the actual stub to call remote synapses. As an Autograd differentiable function, the forward call passes tensors, while the backward pass computes the gradients to get the partial derivative of the remote synapses' output.

forward (ctx, RemoteSynapse, dummy, inputs, mode)

This call passes tensors to the remote synapse. Note that it also takes a `dummy` tensor to prevent torch from excluding the synapse from backward if no other inputs require grad. This method first serializes the inputs, sends them across the wire, then deserializes the outputs of the remote synapses.

Inputs:

- `ctx` : Object that can be used to stash information for backward computation.
- `RemoteSynapse caller` : The calling synapse of type `RemoteSynapse`.
- `torch.Tensor dummy` : A dummy tensor to trigger autograd in the remote synapse.
- `List[ object ] x` : A list of objects to be sent through gRPC to the target remote synapses.
- `bittensor_pb2.Modality mode` : Modality of the data being sent (i.e. Text, Image, etc.)

Outputs:

- The response of each remote synapse. Given that each minibatch runs through the remote synapse's local model, the output back is a vector output of that remote synapse's `forward()` call.

backward (ctx, grads)

The backward function pass computes the gradients of the remote synapse's outputs to get their partial derivative (that is, the gradients). However, at the moment this function does not apply the gradients to the Synapse. Note that if the modality is text, then the derivative would be 0 and thus it returns a `none` tuple. However if the modality is `Image`, `Tensor`, or otherwise, then the gradients of the remote synapses are returned.

Inputs:

- `ctx` : Object that can be used to stash information for backward computation.
- `torch.Tensor grads` : gradients of the local neuron.

Outputs:

- The response of each remote synapse.

7.3 Metagraph class

The Metagraph class is responsible for allowing Neurons to connect to each other and – later – to the blockchain, however this is still under development.

7.3.1 __init__(self, config)

Initializes a new metagraph proof-of-work object cache, initializes gRPC metagraph servicer, and loads config or default configs.

7.3.2 start (self)

Starts two threads:

1. The gossip thread that will communicate with and find potential peers on the network. Target method `_update()`.

2. The gRPC server thread.

7.3.3 update(self)

This keeps the metagraph record of the local neuron up to date by discovering peers using gossip protocol and removing stale peers.

7.3.4 stop (self)

Stops the gossip thread. This is typically called at the end of Bittensor's execution cycle as part of cleanup.

7.3.5 do_gossip (self)

Sends a gossip query to a random peer and records their response into the metagraph via a `sink()` call.

7.3.6 do_clean (self, ttl)

Checks whether a peer hasn't sent a heart beat signal in a given time (i.e. timed out). If it hasn't, then the neuron deletes it from its list of peers.

Inputs:

- `Integer ttl`: Time to live for a given target neuron (defaults to 5 minutes).

Outputs:

- `None`

7.3.7 _sink (self, request)

Records a peer's gossip request to the local neurons list of synapses and peers, and sets its latest heartbeat time.

7.3.8 get_peers(self, n)

Returns the number of active peer endpoints in the network.

Inputs:

- `Integer n` : maximum number of peers to return. This defaults to 10 peers.

Outputs:

- `None`

7.3.9 get_synapses(self, n)

Returns the number of active synapses in the network.

Inputs:

- `Integer n` : maximum number of synapses to return. This defaults to 1000 synapses.

Outputs:

- `None`

7.3.10 `getweights(self, synapses)`

Retrieves the weights for a list of synapses in the network.

Inputs:

- `List[bittensor_pb2.Synapse] synapses` : List of remote synapse endpoints.

Outputs:

- `List(Float) results` : List of remote synapse weights.

7.3.11 `setweights(self, synapses)`

Sets the weights for a list of synapses in the network.

Inputs:

- `List[bittensor_pb2.Synapse] synapses` : List of remote synapse endpoints.
- `torch.Tensor weights` : Weights to set to each synapse.

Outputs:

- `None`

7.3.12 `subscribe(self, synapse)`

Subscribes a synapse class object to the neuron's metagraph.

Inputs:

- `bittensor_pb2.Synapse synapses` : Synapse to subscribe to metagraph.

Outputs:

- `None`

7.4 Synapse class

This is the **superclass** of every custom deep learning model that users will write on Bittensor. More specifically, each deep learning model (read: neuron, or miner) will extend this class to gain the capabilities of communicating with other bittensor neurons.

7.4.1 `__init__ (self)`

Initializes the *Synapse public key* and identifies whether the current machine is cuda-capable (i.e. can we train on the GPU?).

7.4.2 forward_text (self), forward_image (self), and forward_tensor (self)

These methods are not implemented in this class, but are defined as contractual methods where one (or all) of them should be implemented by the user and their model they are building (See examples like MNIST).

Inputs:

- `torch.Tensor inputs` : The batch of data to be sent through the `forward()` call of the custom model.

7.4.3 call_forward (self, inputs, modality)

Apply forward pass to the bittensor.synapse given inputs and modality. The modality passed to this function determines whether it calls `forward_text (self)`, `forward_image (self)`, or `forward_tensor (self)`. Therefore, one of them **must** be implemented in the subclass (custom) model.

Inputs:

- `Object inputs` : The batch of data to be sent through the `forward()` call of the custom model.
- `modality` : The modality of the data being sent.

Outputs:

- `torch.Tensor Outputs` : The output of the `forward()` call of the model.

7.4.4 call_backward (self, inputs, grads)

Apply a backward pass to the synapse given grads and inputs. Presently, this only returns a tensor of zeros as it is not being used.

Inputs:

- `Object inputs` : The batch of data to be sent through the `forward()` call of the custom model.
- `grads` : Gradients of the remote synapses.

Outputs:

- `torch.zeros((1,1))` : A tensor of zeros.