PyTorchPipe

A framework for rapid prototyping and training of computational pipelines combining language and vision

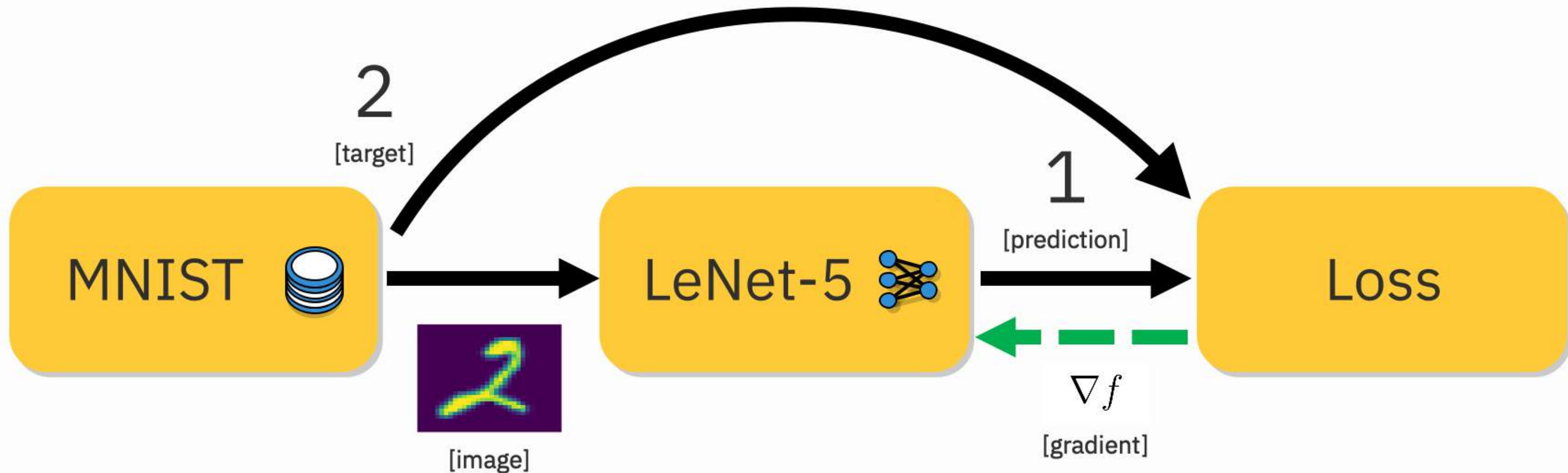
Tomasz Kornuta
Machine Intelligence Team
Almaden Research Center
IBM Research AI

Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

Story 1

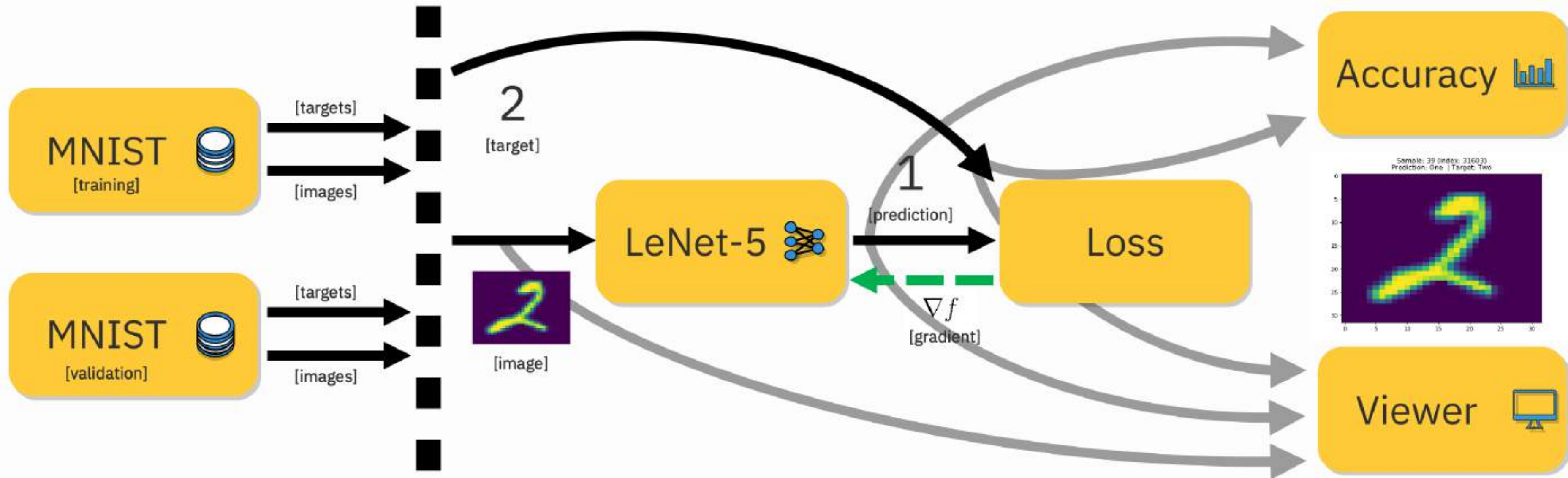
- **What do you do when you want to train a model**?
  - pick your favorite *middleware* (e.g. PyTorch)
  - pick a *dataset* (e.g. MNIST) and a *model* (e.g. LeNet-5)



- **coding, coding…**

Story 2

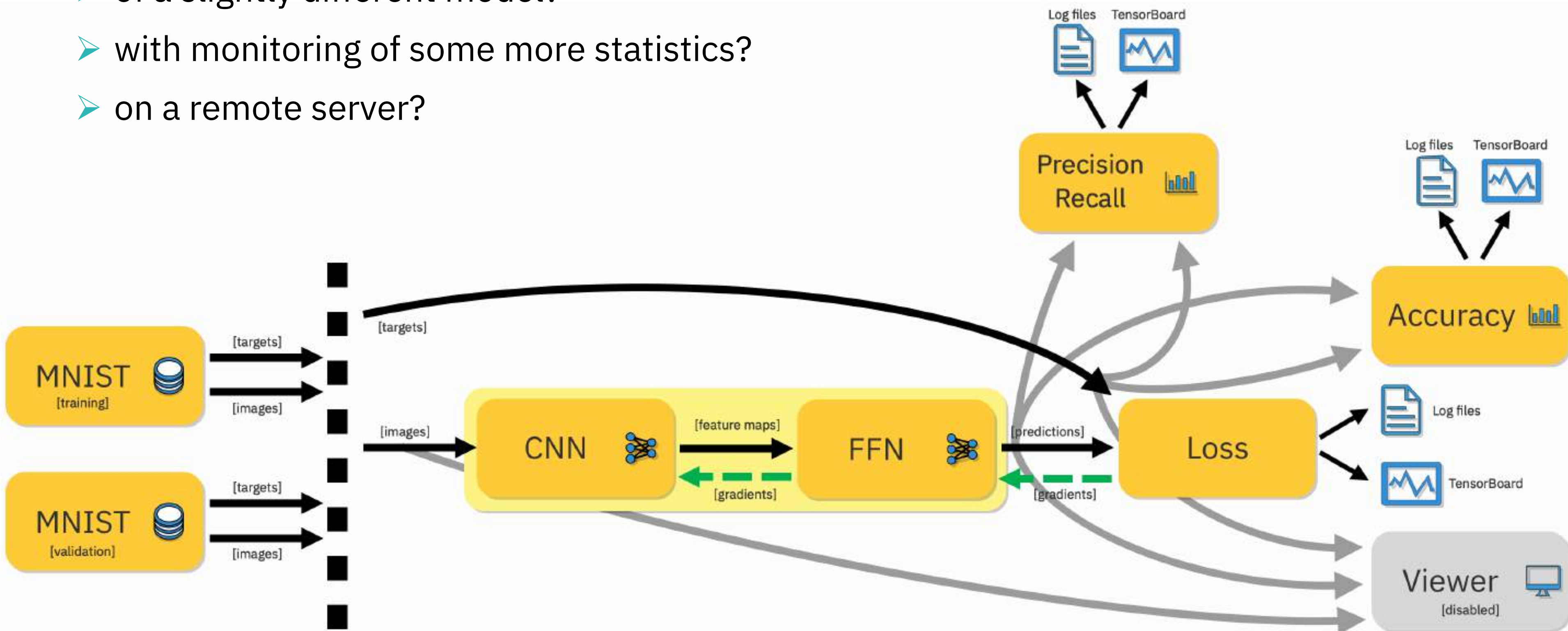
- **How about training with…**
  - dataset split into training and validation?
  - monitoring of accuracy?
  - visualization of results?



- **coding, coding…**

Story 3

- **And how about training…**
  - of a slightly different model?
  - with monitoring of some more statistics?
  - on a remote server?



- **some more coding…**

Story 4

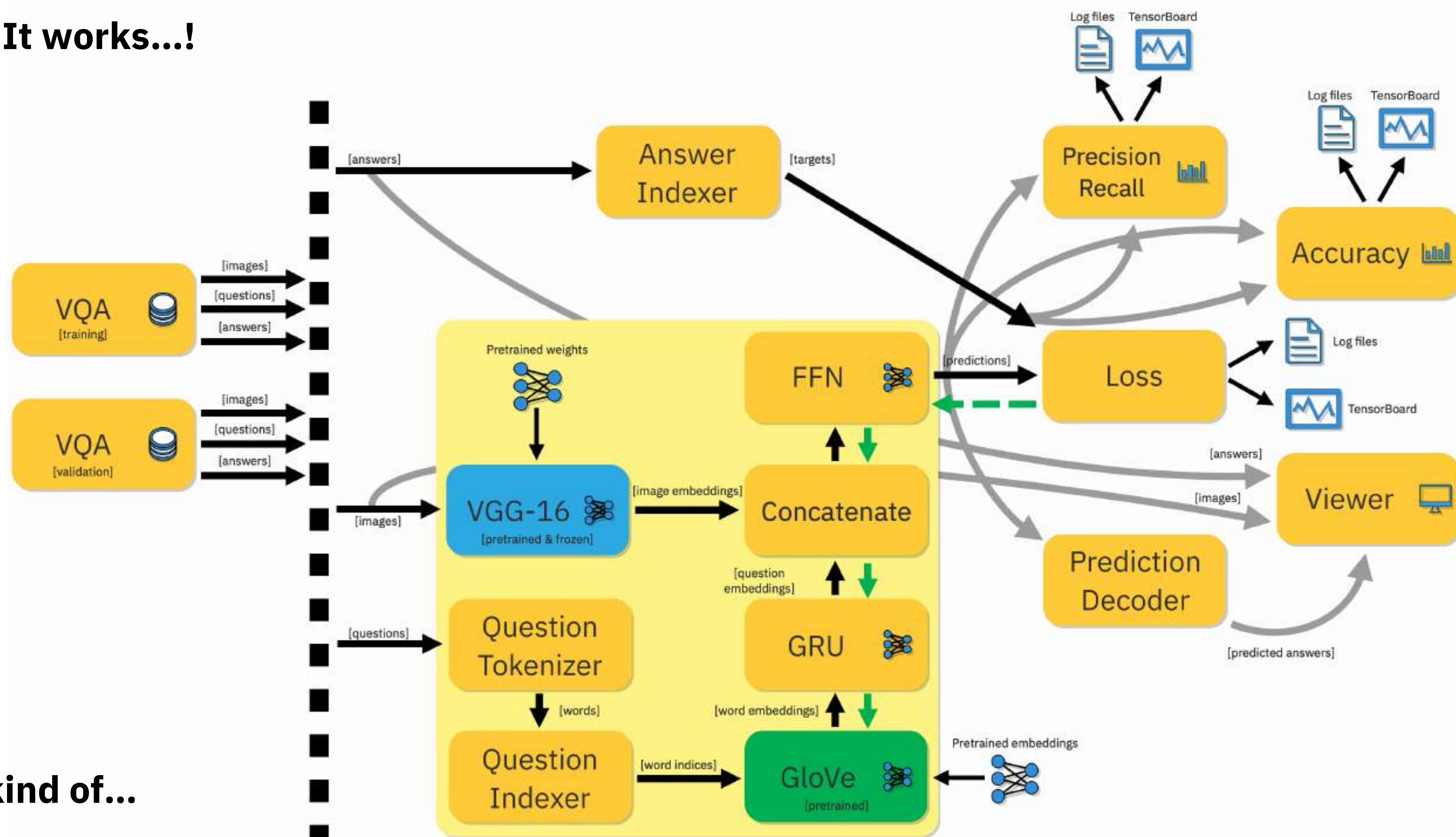
- **Next you move to the Visual Question Answering problem domain and...**

  - copy some code snippets from your previous solutions
  - decide to implement a simple multi-modal fusion (e.g. concatenation)
  - need to incorporate some pretrained word embeddings (e.g. GloVe)
  - need to incorporate a pretrained image encoder (e.g. VGG-16)
  - ... and need to solve all other unexpected (but encountered) issues!

- **coding, debugging, training, coding, coding, coding, debugging, training, debugging...**

Story 5

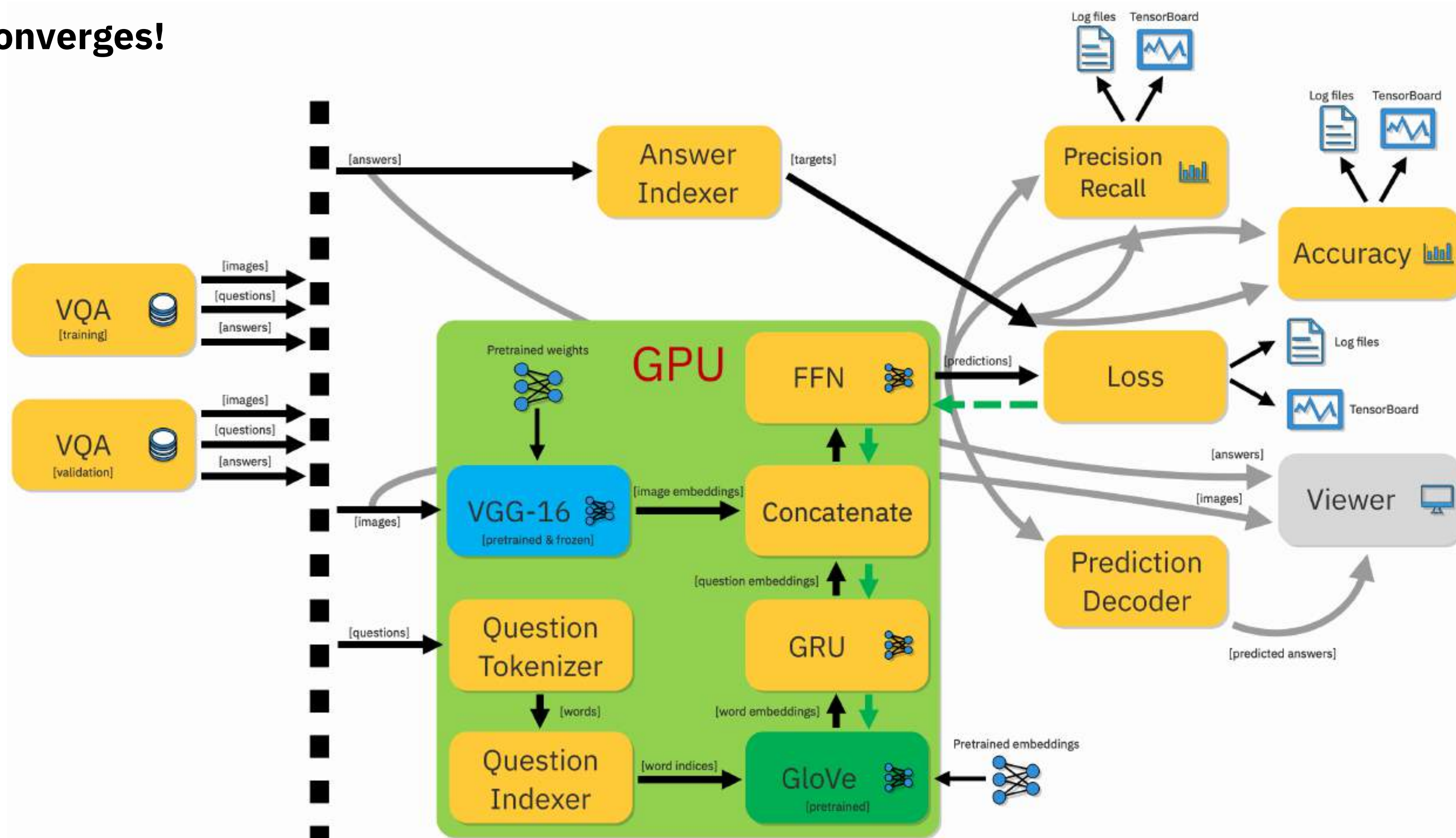
- **Phew! It works...!**



- **Well, kind of...**
  - It takes ages to train!
  - Must use GPUs!
- **Coding...**

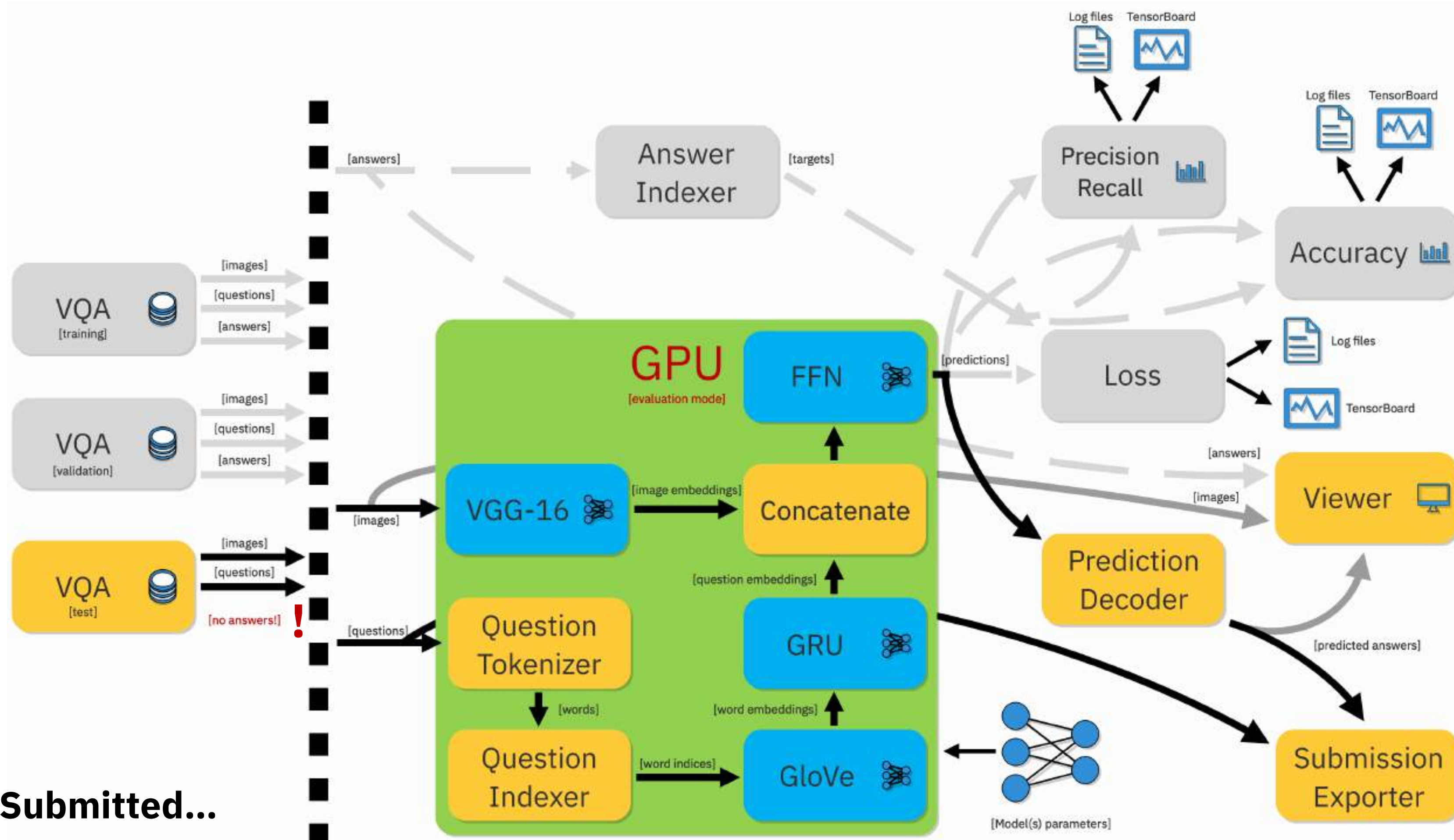
Story 6

- **Yay! Converges!**



- **Now: the test scores...**
  - (+ saving the model)
- **Coding...**

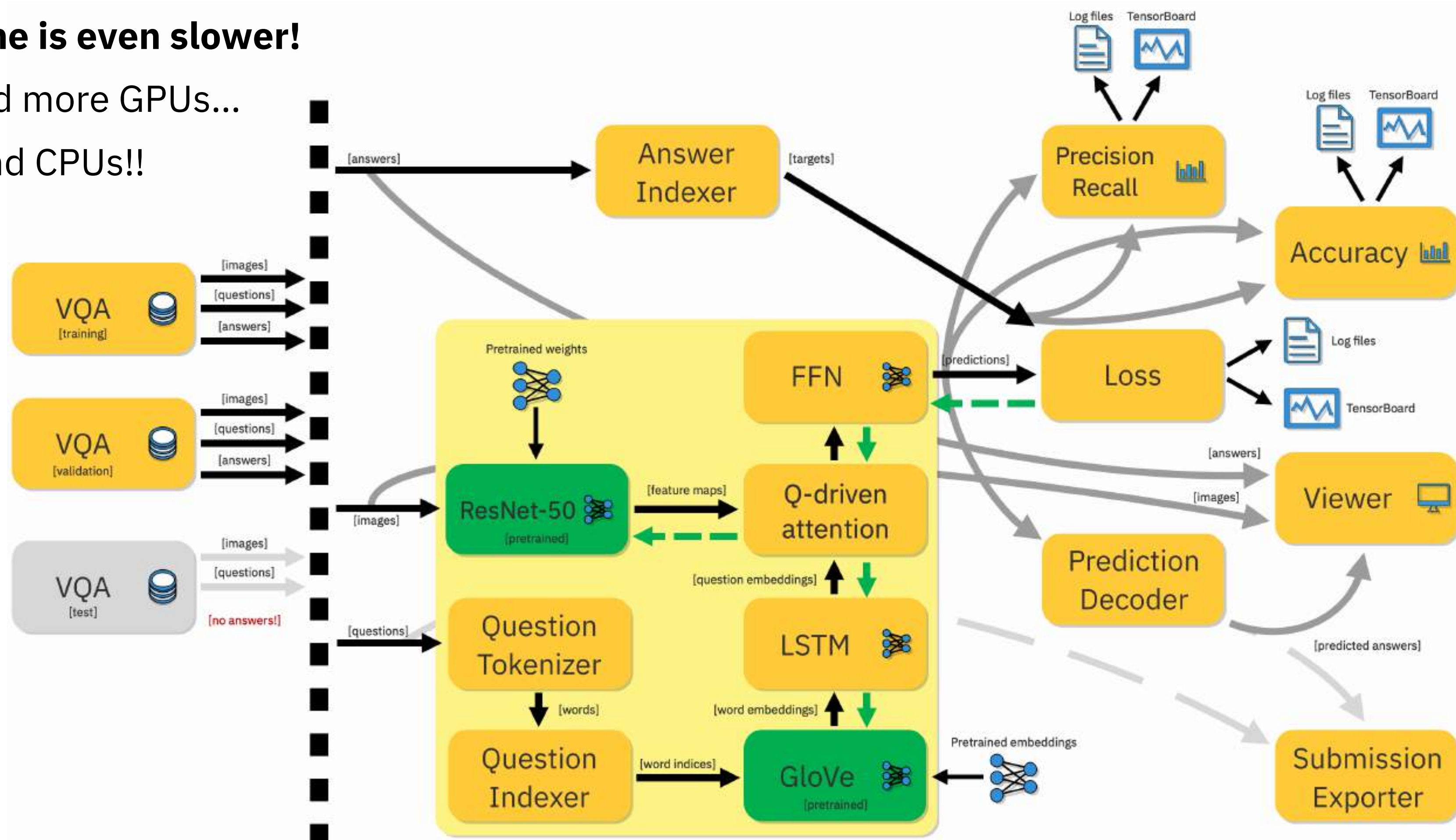
Story 7



- **Got it! Submitted…**
  - But the scores are so low…
  - Need a better model!
- **Coding…**

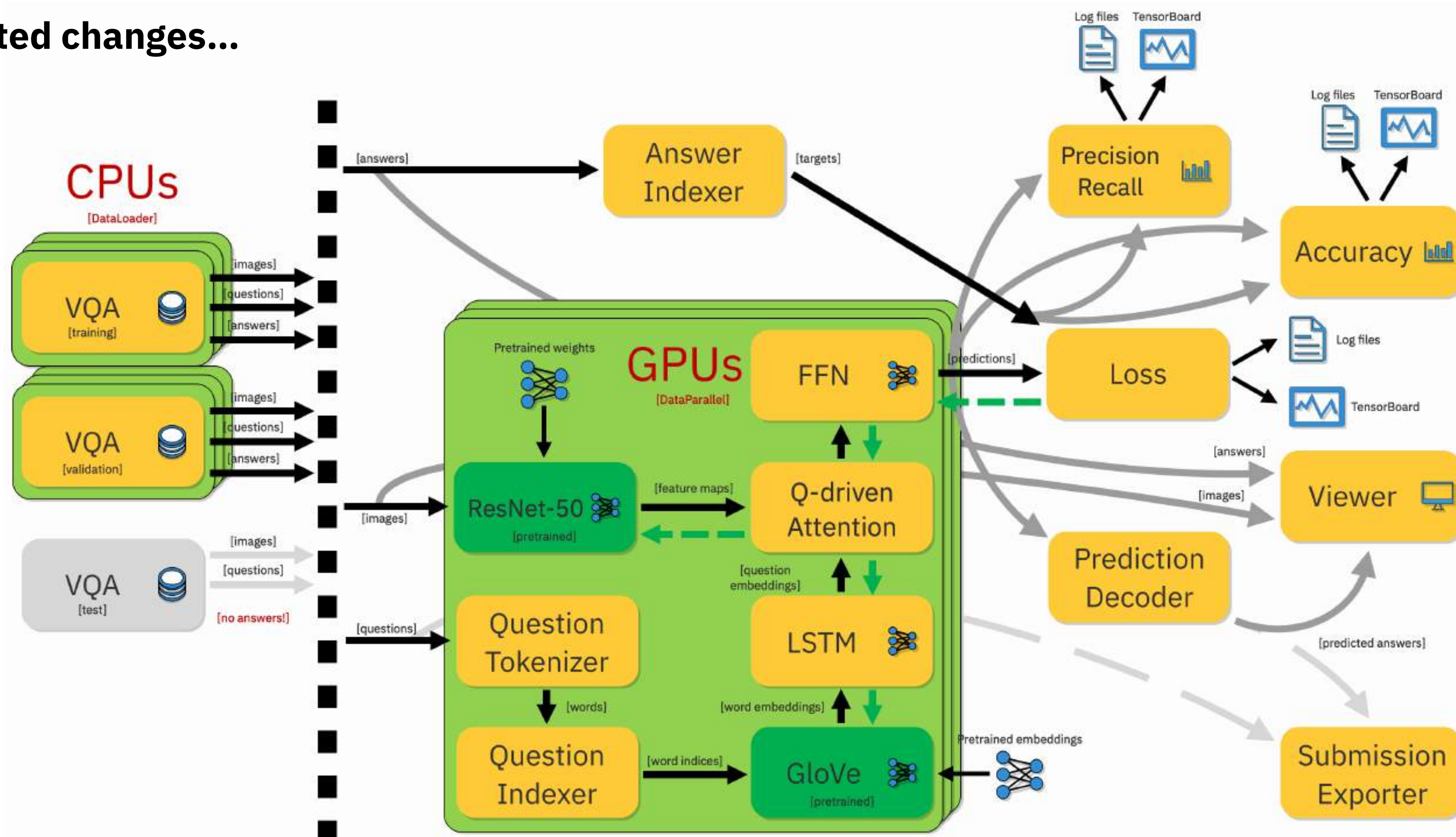
Story 8

- **This one is even slower!**
  - Need more GPUs…
  - … and CPUs!!



Story 9

- **Reverted changes…**



- **Training…**
  - Now I need to run tests once again… ouch!

Story summary: the *requirements*

- “Plugging” in/out “modules” realizing different functionalities

- Importing the pretrained models (or their “parts”)…
  - (… and saving them after the training)

- Freezing/unfreezing the models (or their “parts”) at run-time

- Disabling some of the “modules” at run-time

- Run-time parametrization of all “modules” (hyperparameters!)…
  - .. and training/test procedures!

- Utilization of many CPUs/GPUs (on demand at run-time)

- … logging, statistics collection, export to files (e.g. to TensorBoard), visualization…

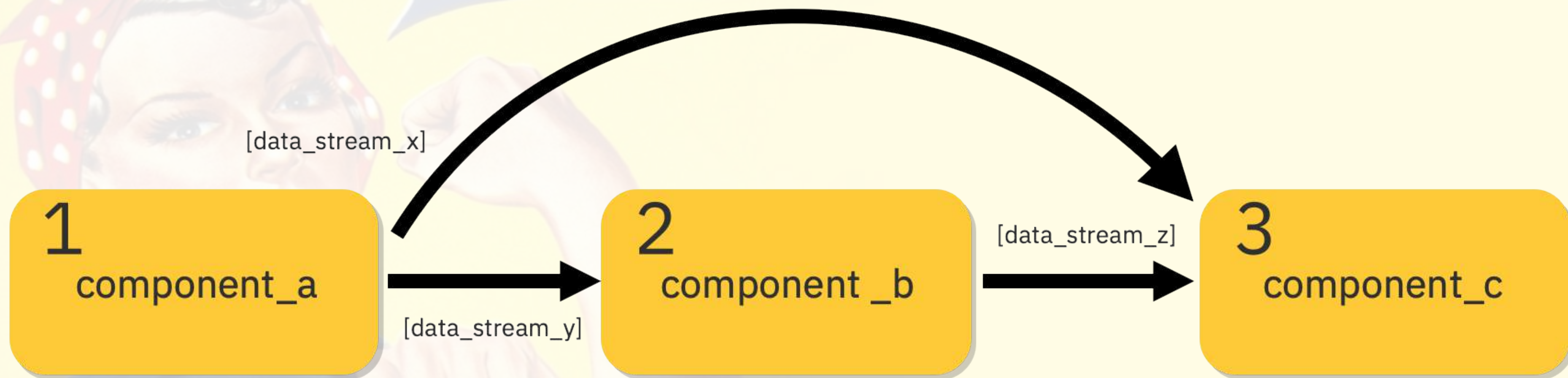
Ok, can we have a tool facilitating all that?

Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

Welcome to PyTorchPipe! (or PTP in short)

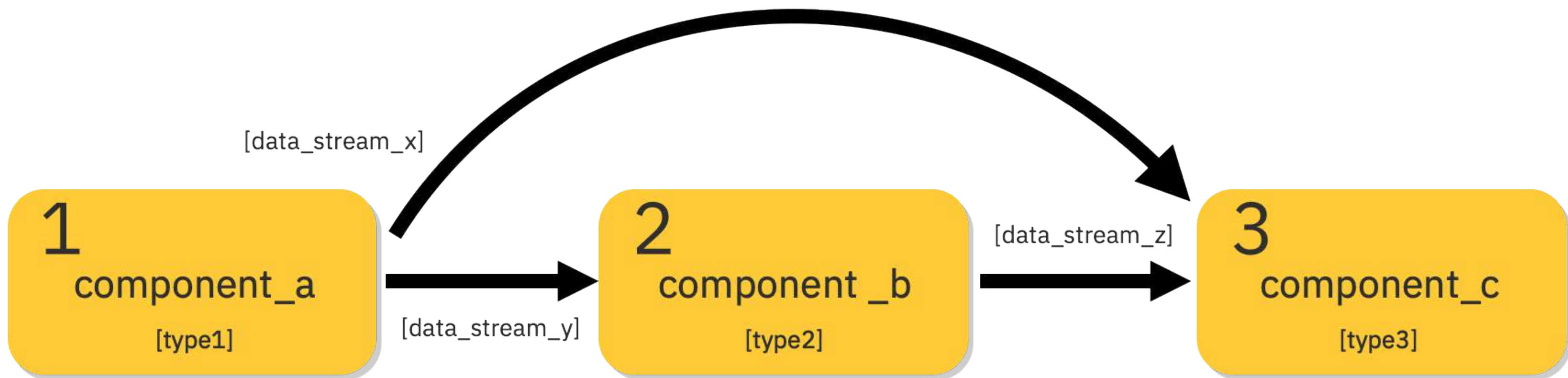
- **Pipeline": formalization of a concept of a "high-level computational graph" (DAG)**
  - *Components* connected via *data streams and* executed by their *priority*



➢ **Yes, we can!**

PTP: pipeline 1

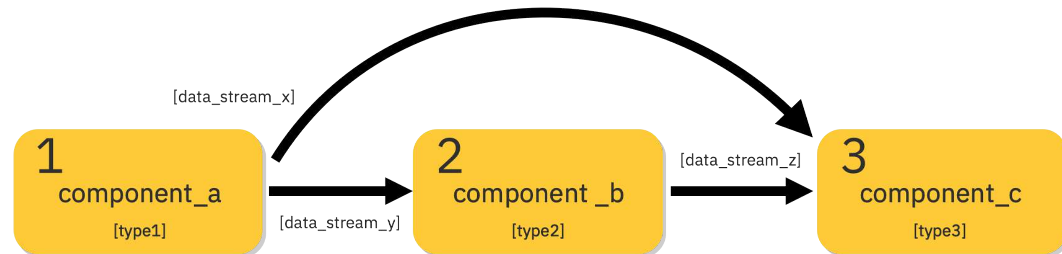
- Components are unaware of each other (loose coupling)
- Components only care about their *input* and *output (data) streams*
- Components have different types



PTP: pipeline 2

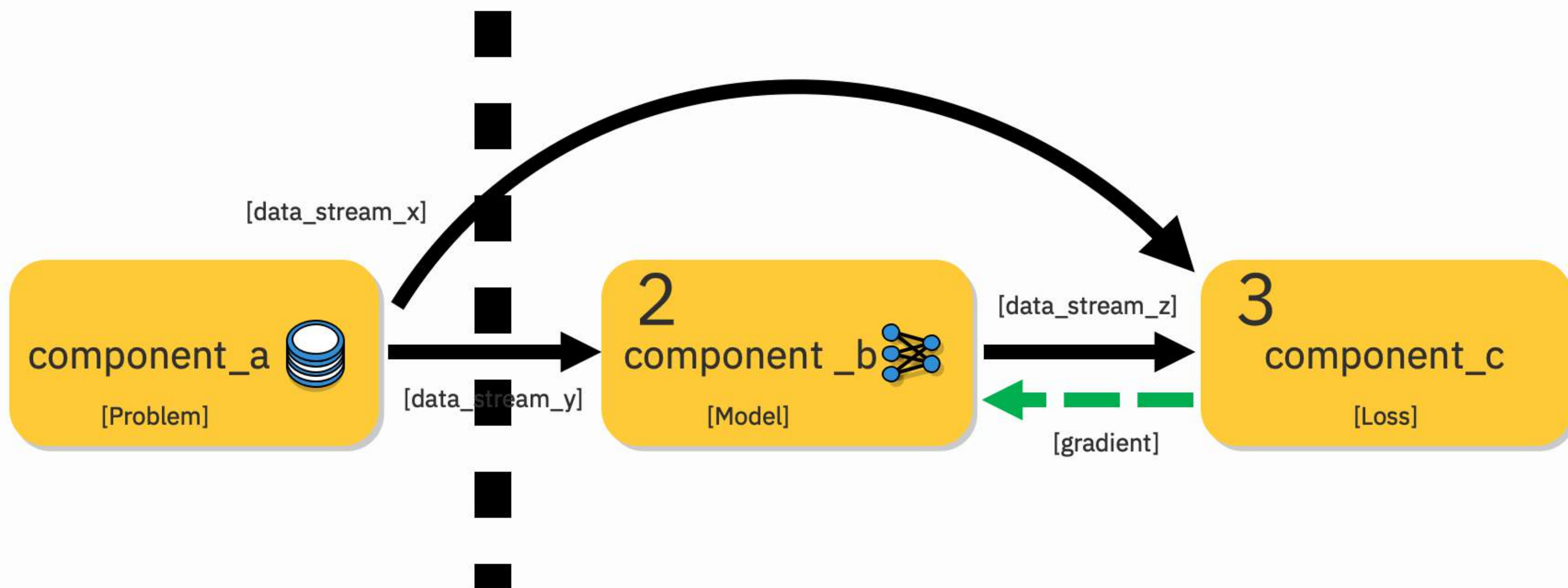
- **Pipelines** are defined in **.yml** files

```
pipeline:
    component_a:
        priority: 1
        type: type1
        streams:
            output1: data_stream_x
            output2: data_stream_y

    component_b:
        priority: 2
        type: type2
        streams:
            input: data_stream_y
            output: data_stream_z

    component_c:
        priority: 3
        type: type3
        streams:
            input1: data_stream_x
            input2: data_stream_z
```



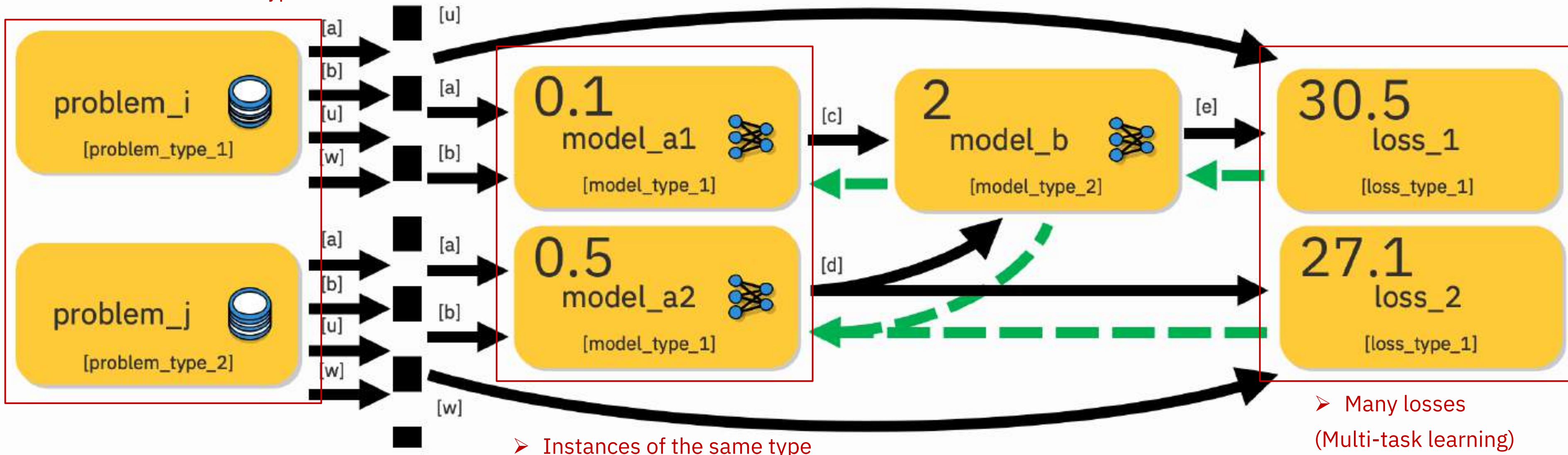
PTP: pipeline 3

- Three *special* types of components: *Task, Model, Loss*
- *Tasks* are treated in a *special* way (thus outside of *pipeline*).. but are still components!



PTP: pipeline 4

- **Everything is allowed…**



- **… as long as:**
  - Names are unique,
  - priorities make sense and
  - *data stream* types fit!

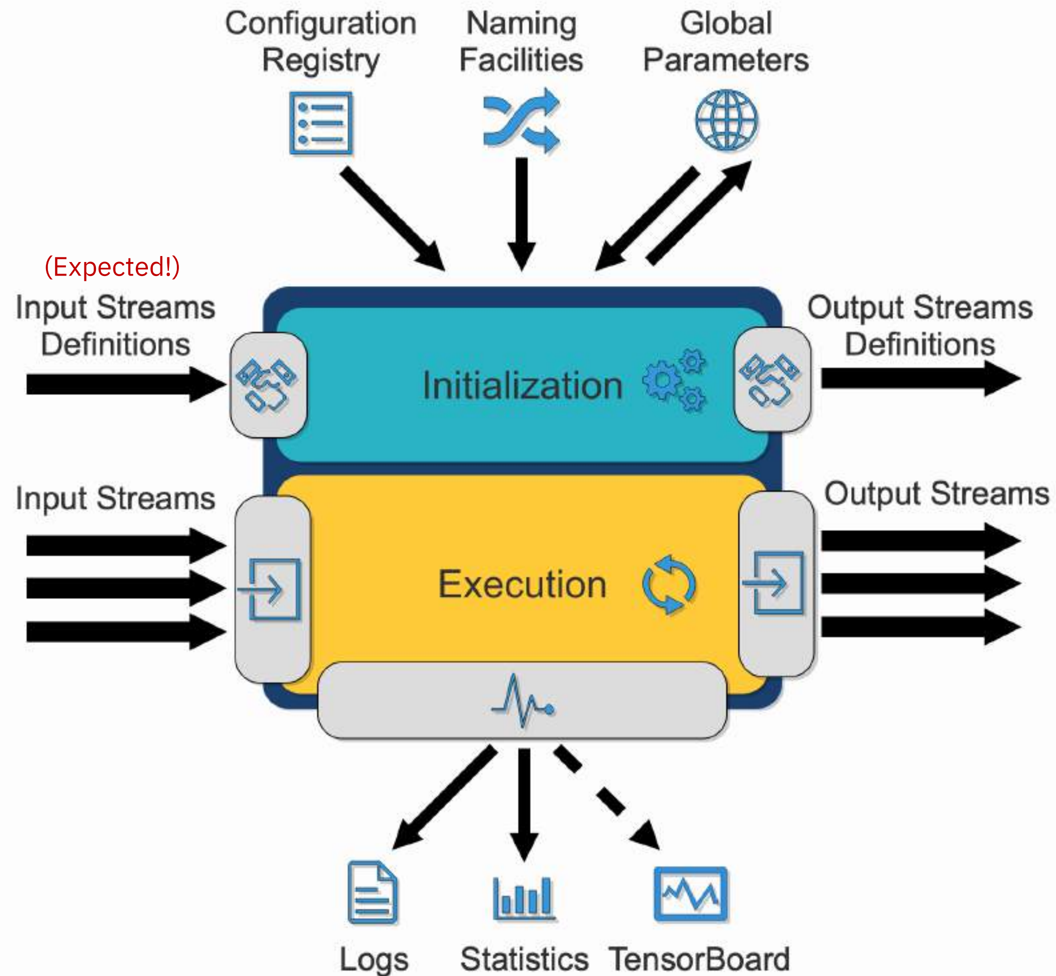
Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

PTP: component 1

- **Initialization**
  - Load configuration from .yml files
  - Rename stream/global names

```yaml
pipeline:
  # LeNet-5 model.
  image_classifier:
    priority: 1
    type: LeNet5
    streams:
      inputs: images
      predictions: lenet5_predictions
    globals:
      prediction_size: num_classes
```



  - Get/set global variables
  - Handshake output-input definitions

PTP: component 2

- **Example:** the LeNet5 model default configuration file (lenet5.yml)

```yaml
# This file defines the default values for the LeNet5 model.
####################################################################
# 1. CONFIGURATION PARAMETERS that will be LOADED by the component.
####################################################################

streams:
     ####################################################################
     # 2. Keymappings associated with INPUT and OUTPUT streams.
     ####################################################################

     # Stream containing batch of images (INPUT)
     inputs: inputs

     # Stream containing predictions (OUTPUT)
     predictions: predictions

globals:
     ####################################################################
     # 3. Keymappings of variables that will be RETRIEVED from GLOBALS.
     ####################################################################

     # Size of the prediction (RETRIEVED)
     prediction_size: prediction_size
```

Note: All components come with files defining their *default configurations*. Check them out!

PTP: component 3

- **Example:** the LeNet5 model initialization (lenet5.py)

```
class LeNet5(Model):
    def __init__(self, name, config):
        super(LeNet5, self).__init__(name, LeNet5, config)

        # Get key mappings.
        self.key_inputs = self.stream_keys["inputs"]
        self.key_predictions = self.stream_keys["predictions"]

        # Retrieve prediction size from globals.
        self.prediction_size = self.globals["prediction_size"]

        # Create the LeNet-5 layers.
        self.conv1 = torch.nn.Conv2d(1, 6, kernel_size=(5, 5))
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        self.conv2 = torch.nn.Conv2d(6, 16, kernel_size=(5, 5))
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        self.conv3 = torch.nn.Conv2d(16, 120, kernel_size=(5, 5))
        self.linear1 = torch.nn.Linear(120, 84)
        self.linear2 = torch.nn.Linear(84, self.prediction_size)
```

PTP: component 4

- **Example:** the LeNet5 model input/output data streams definitions

  - Components can have many input/output data streams, each need a definition!
  - Data stream definition is a triplet: sizes, types and description
    - -1 is a special value, meaning “can work with different sizes”

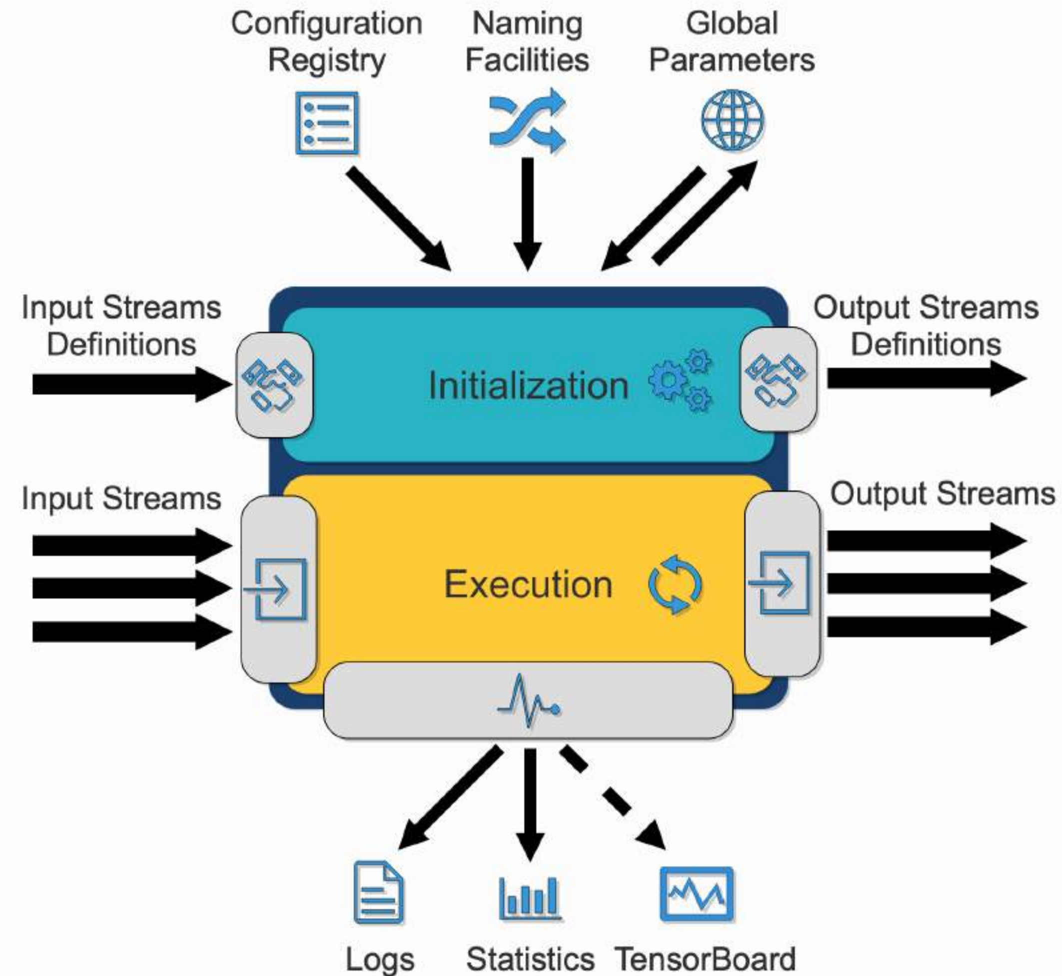
```
def input_data_definitions(self):
  return {
    self.key_inputs: DataDefinition([-1, 1, 32, 32], [torch.Tensor],
      "Batch of images [BATCH_SIZE x IMAGE_DEPTH x IMAGE_HEIGHT x IMAGE WIDTH]"),
    }


def output_data_definitions(self):
  return {
    self.key_predictions: DataDefinition([-1, self.prediction_size], [torch.Tensor],
      "Batch of predictions, each represented as probability distribution over classes
[BATCH_SIZE x PREDICTION_SIZE]")
    }
```

Note: Tasks are sources of data, so need only output data definitions!

PTP: component 5

- **Execution**
  - Process inputs into outputs

```
def forward(self, data_streams):
  # Get image from data streams.
  img = data_streams[self.key_inputs]

  # Pass inputs through layers.
  x = self.conv1(img)

  # …

  # Log softmax.
  predictions = f.log_softmax(x, dim=1)
  # Add predictions to data streams.
  data_streams.publish(
    {self.key_predictions: predictions})
```



  - Logging facilities
  - Statistics collectors/aggregators

Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

PTP provides… the task zoo

- **"Task":** a non-trainable component fetching data to the pipeline

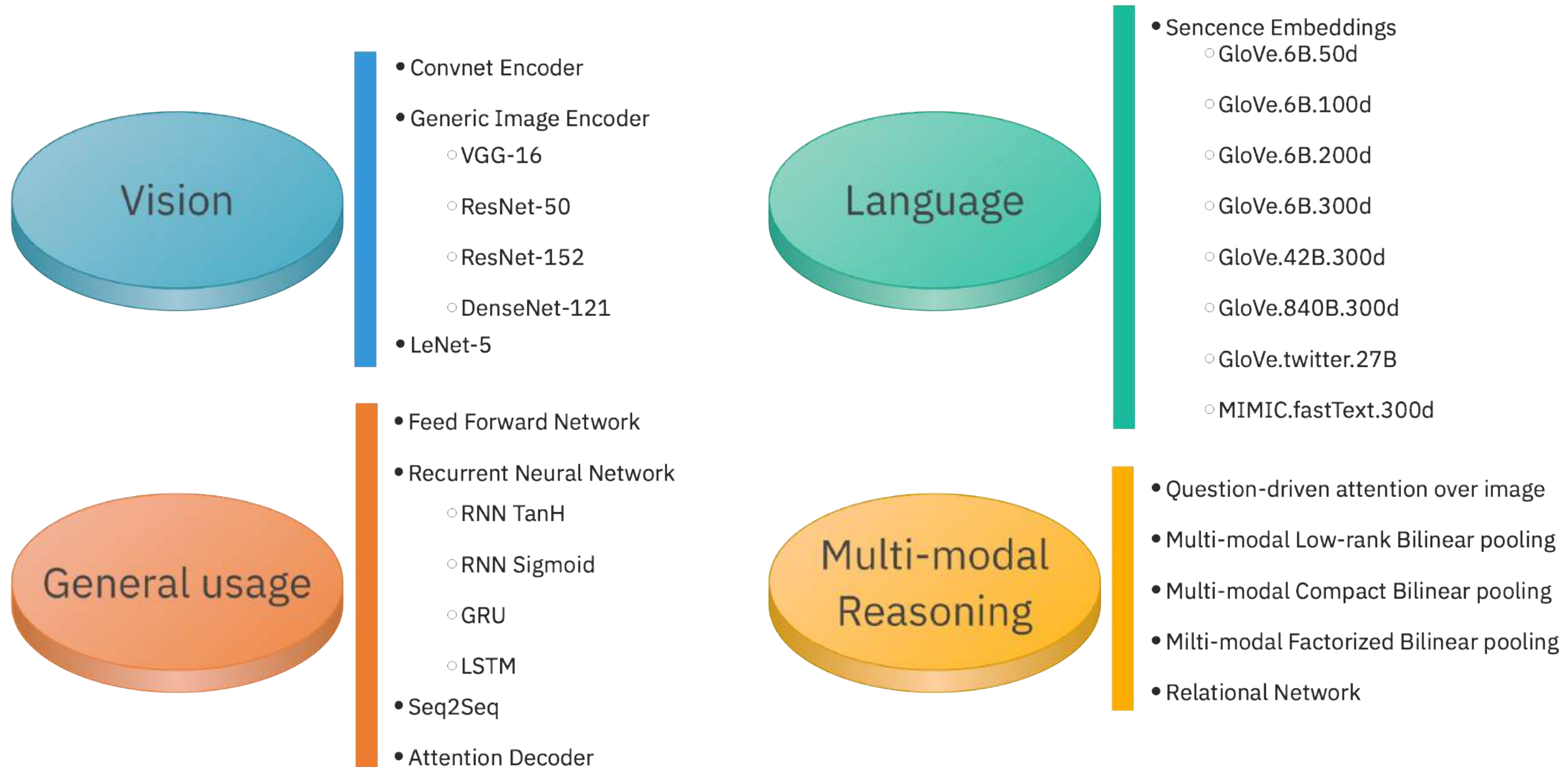
Vision
- MNIST (Image Classification)
- CIFAR-100 (Image Classification)

Language
- WiLY (Language Identification)
- WikiText-2 / WikiText-103 (Language Modelling)
- ANKI (Machine Translation)

Multi-modal
- CLEVR (Visual Question Answering)
- GQA (Visual Question Answering)
- ImageCLEF VQA-Med 2019 (Visual Question Answering)

PTP provides... the model zoo

- **“Model”:** a component with trainable weights



PTP provides...

- **Other components**

- Sentence Tokenizer
- Sentence Indexer
- Sentence One Hot Encoder
- Label Indexer
- BoW Encoder
- Word Decoder

- Accuracy Statistics
- BLEU Statistics
- Batch Size Statistics
- Precision/Recall Statistics
- NLL Loss ➢ Multi-task learning!

- Global Variable Publisher
- Image Viewer
- Stream Viewer
- Stream File Exporter

- List to Tensor
- Reshape Tensor
- Reduce Tensor
- Concatenate Tensors

PTP provides…. workers

- **ptp-offline-trainer**
  - A trainer relying on classical methodology interlacing training and validation at the end of every epoch. Creates separate instances of training and validation tasks and trains the models by feeding the created pipeline with batches of data.

- **ptp-online-trainer**
  - A flexible trainer creating separate instances of training and validation tasks and training the models by feeding the pipeline with training data. Validation is performed on as subset of the validation set and user might set how often it is executed.

- **ptp-processor**
  - Worker performing one pass over the all samples returned by a given task instance, useful for collecting scores on a given set, answers for submissions to competitions etc.

Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

How to use workers

1) PTP offers three general-usage *workers* scripts

- **Not sure how to use them?**
  - Simply call given worker with **–h** option to learn about its run-time arguments

```
foo@bar:~$ ptp-offline-trainer --h
```

  - Each worker comes with *default configuration file* located in *configs/default/workers/*

```
####################################################################
# Section defining all the default values of parameters used during training when using ptp-offline-trainer.
# If you want to use different section for "training" pass its name as command line argument '--
training_section_name' to trainer (DEFAULT: training)
# Note: the following parameters will be (anyway) used as default values.
default_training:
      # Set the random seeds: -1 means that they will be picked randomly.
        # Note: their final values will be stored in the final training_configuration.yml saved to log dir.
        seed_numpy: -1
        seed_torch: -1

      # Default batch size.
        batch_size: 64

      # Definition of the task (Mandatory!)
        #task:
```

How to use components

2) PTP comes with component/model zoo

- **Not sure what are component global variables, data streams etc.**
  – Open the associated *default configuration file* located in *configs/default/components/…*

```
####################################################################
# 1. CONFIGURATION PARAMETERS that will be LOADED by the component.
####################################################################

# Folder where task will store data (LOADED)
data_folder: '~/data/mnist'

# Defines the set that will be used used (LOADED)
# True: training set | False: test set.
use_train_data: True

# Optional parameter (LOADED)
# When present, resizes the MNIST images from [28,28] to [width, height]
#resize_image: [height, width]

streams:
        ####################################################################
        # 2. Keymappings associated with INPUT and OUTPUT streams.
        ####################################################################

        # Stream containing batch of indices (OUTPUT)
        indices: indices
```

- **Not sure what given component does?**
  ➢ If anything else fails… simply open the source file ;)

How to build pipelines 1

3) Rapid prototyping!

- Compose your pipeline from the existing library of components…
  - PTP will **assist you** in that task!

- **Not sure whether you connected your components correctly?**
  - No problem! PTP will check your pipeline by handshaking output-input definitions of streams of all component and will informing you if something is improper, e.g.:

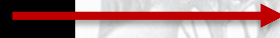
```
[2019-06-11 20:37:05] - ERROR - reshaper >>> Input definition: expected field 'feature_maps1' not found in
DataStream keys (stream_keys(['indices', 'inputs', 'targets', 'labels', 'feature_maps']))
```

```
[2019-06-11 20:38:51] - ERROR - reshaper >>> Input definition: field 'inputs' in DataStream has dimension 3
different from expected (expected [-1, 16, 1, 1] while received [-1, 1, 28, 28])

[2019-06-11 20:38:51] - ERROR - OfflineTrainer >>> Found 6 errors, terminating execution
```

How to build pipelines 2

- **Your configuration file is getting too complex?**
  - No problem! Use the build-in mechanism for **nesting** of the configuration files!

```
# Load config defining MNIST tasks for training,
validation and testing.
default_configs: mnist/default_mnist.yml
# …
```



```
training:
	task:
		type: MNIST
		# …
```

- **Got conflicts with names of data streams/global variables?**
  - No problem! PTP has built-in (re)**naming facilities**!

```
classifier:
	priority: 3
	type: FeedForwardNetwork
	# Rename stream names.
	streams:
		inputs: reshaped_maps
	# Rename global variable names.
	globals:
		input_size: reshaped_maps_size
		prediction_size: num_classes
```

Note: All components come with files defining their *default configurations*. Check them out!

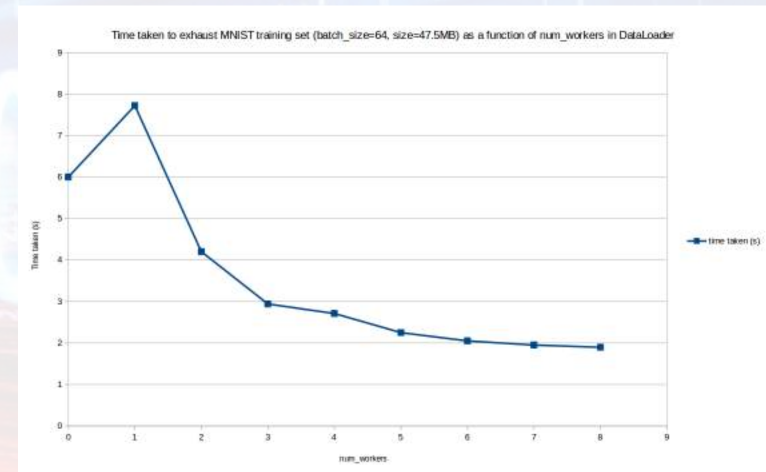
How to use CPUs and GPUs 1

4) Effort-less harnessing of the computational power!

- Want to many **CPU cores/processes** for loading data when creating e.g. *training* batches?

  - Simply add to the *training* section in your configuration file the following:

```
training:
    # …
    # Use four workers for loading data.
    dataloader:
        num_workers: 4
```



- **Note:** All workers come with files defining their *default configurations*. Check them out!

How to use CPUs and GPUs 2

- Want to use **many GPUs** for forward and backward batch propagation?

  - Simply call a given worker script with **–gpu** flag on, e.g.:

```
foo@bar:~$ CUDA_VISIBLE_DEVICES=0,1 ptp-processor … --gpu
```

Note: Without setting the environment variable worker with –gpu will use **all GPUs** by default!

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX TIT...  On   | 00000000:04:00.0 Off |                  N/A |
| 22%   50C    P2    74W / 250W |    571MiB / 12212MiB |     23%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX TIT...  On   | 00000000:05:00.0 Off |                  N/A |
| 22%   46C    P2    73W / 250W |    558MiB / 12212MiB |      8%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX TIT...  On   | 00000000:09:00.0 Off |                  N/A |
| 22%   46C    P2    72W / 250W |    558MiB / 12212MiB |      8%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX TIT...  On   | 00000000:83:00.0 Off |                  N/A |
| 22%   48C    P2    72W / 250W |    558MiB / 12212MiB |      8%      Default |
+-------------------------------+----------------------+----------------------+
|   4  GeForce GTX TIT...  On   | 00000000:84:00.0 Off |                  N/A |
| 22%   45C    P2    72W / 250W |    558MiB / 12212MiB |      8%      Default |
+-------------------------------+----------------------+----------------------+
|   5  GeForce GTX TIT...  On   | 00000000:87:00.0 Off |                  N/A |
| 22%   45C    P2    72W / 250W |    558MiB / 12212MiB |      7%      Default |
+-------------------------------+----------------------+----------------------+
|   6  GeForce GTX TIT...  On   | 00000000:88:00.0 Off |                  N/A |
| 22%   47C    P2    75W / 250W |    554MiB / 12212MiB |      6%      Default |
+-------------------------------+----------------------+----------------------+
```

- We encourage utilization of the **cuda-gpupick** tool for automatic *optimal* allocation on GPUs

```
foo@bar:~$ cuda-gpupick -n2 ptp-processor … --gpu
```

Agenda

- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

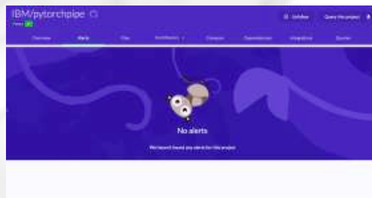
How to develop components 1

1) Focus of attention!

- Compose your pipeline from the existing library of components…
  - …and focus on developing your new component

```
def forward(self, data_streams):

      # Get image from data streams.
      img = data_streams[self.key_inputs]

      # Pass inputs through layers.
      x = self.conv1(img)
      # …
```

  - … then use the existing *worker* scripts to test it!
- PTP supports **Test-Driven Development**! Write unit tests first and let **DevOps** do their job!



How to develop components 2

2) Parametrization

- Derive your component by deriving it from adequate parent class
  - *Task, Model, Loss, Component* are four major base classes

- Divide parameters of your component into:
  - Parameters **loaded** at run-time from the **configuration**
    - i.e. internal parameters that do not depend on other components
    - Example: *FeedForwardNetwork* loads number of hidden layers along with respective numbers of neuron in each of them
  - Parameters **retrieved** at run-time from **global variables**
    - i.e. parameters set by other components
    - Example: *AnswerDecoder* retrieves word mappings created by other components
  - Parameters **exported** to **global variables**
    - i.e. parameters that other components might find useful
    - Example: *MNIST* exports number of target classes

How to develop components 3

3) Implementation

- Define input and output streams
  - Implement methods returning input/output stream definitions

Good practice: Get stream keys using naming facilities in the *init()* method

- Implement the *forward()* method realizing the main operation

Note: Task classes do not use forward(), instead require __getitem__() and collate() methods

- Each component comes with default configuration
  - They need to be developed concurrently!

- We strongly encourage Test-Driven Development
  - Start your implementation from writing unit tests, run them as you progress!

```
foo@bar:~$ coverage run -m unittest
```

Agenda

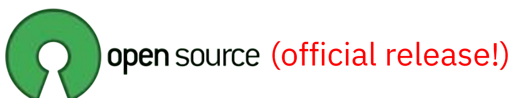
- **Motivation**
- **PyTorchPipe**
  - Pipeline explained
  - Component explained
  - Task/model/component zoo, workers
- **How to**
  - Use workers, components, build pipelines, use CPUs and GPUs
  - Develop components
- **Summary**

PyTorchPipe: summary 1

- **PyTorchPipe** is an open-source framework accelerating AI/Machine Learning research

- It provides datasets, a growing model zoo and predefined pipelines to jump start research on various multi-modal tasks (with focus on, but not limited to, **vision** and **language**).

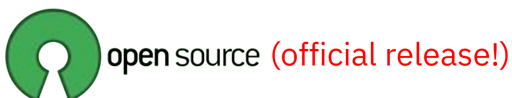
- **Key Features:**
  - Flexible **pipelines** with **plug-and-play modules (components)**
  - Support for **multi-task training** (multiple loss functions)
  - Easy to read **yaml** files describing pipeline and training/test configuration
  - Out-of-the-box **distributed computing** over multiple GPUs/CPUs
  - **Logging** and **visualization** tools



https://github.com/IBM/pytorchpipe language Python | license Apache-2.0 | version 0.1.0 | build passing | code quality: python A+ | lgtm 0 alerts

PyTorchPipe: summary 2

- 3 universal, configurable *workers*: scripts for training/testing of all kinds of pipelines
- 8 configurable *tasks* from diverse domains (visual question answering, image classification, language classification, language modelling, machine translation)
- 32 configurable *components*, including 13 trainable *models*
- around 80 *pipelines*, varying from simple ones with few components for MNIST classification to ones consisting of 50+ components (!) for VQA



(official release!)

https://github.com/IBM/pytorchpipe language Python license Apache-2.0 version 0.1.0 build passing code quality: python A+ lgtm 0 alerts

PyTorchPipe: comparison

	Tensor2Tensor	Ludwig	Pythia	MI-Prometheus	PyTorchPipe
Middleware	TensorFlow	PyTorch	PyTorch	PyTorch	PyTorch
Multi-modal tasks	Yes?	No	Yes	Yes	Yes
Model split into many pieces	No	Encoder-Combiner-Decoder (fixed)	No	No	Yes
Multi-task learning (many losses at once)	Yes?	No	Yes	No	Yes
Configuration files	No	yaml	yaml	yaml	yaml
General scripts / workers	trainer	train, predict, test, experiment, visualize, collect_weights, collect_activations	trainer	2 x trainer, tester	2 x trainer, processor
Support for many GPUs	Yes	Yes	Yes	No (1 GPU max)	Yes
Many CPU workers during data loading	Yes?	MPI + Horovod (external)	Yes	Yes	Yes
Logging	Yes	Yes	Yes	Yes	Yes
Statistics collection	TensorBoard	json	TensorBoard	csv files, TensorBoard	csv files, TensorBoard
Visualization of intermediate results	No	No	No	In models (matplotlib)	Dedicated components (matplotlib)
Visualization after run	TensorBoard	Yes (visualize)	TensorBoard	TensorBoard	TensorBoard
Organization	Google Brain	Uber AI	Facebook AIR	IBM Research AI	IBM Research AI