MPI-based tools for large-scale training and optimization at HPC sites

Vladimir Loncar [1], Jean-Roch Vlimant [2], Sofia Vallecorsa [1], Gul Rukh Khattak [3], Maurizio Pierini [1], Thong Nguyen [2], Federico Carminati [1]

[1] CERN
[2] California Institute of Technology
[3] University of Peshawar

Overview

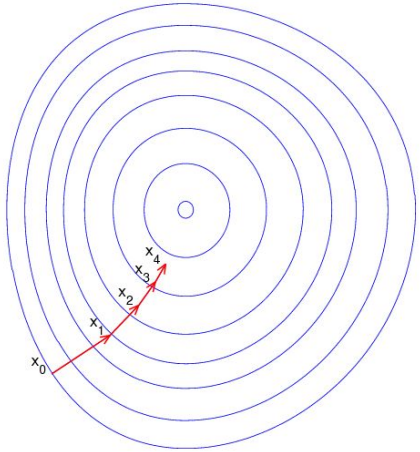
- Challenges in large-scale network training
- Distributed training
- Types of parallelism in distributed training
- Distributed network optimization
- NNLO library
- Performance analysis
- Conclusion

Challenges in large-scale network training

- Machine learning prototyping is easier than ever
  - Many libraries: Keras, PyTorch, Tensorflow...
- Out-of-the-box support for training with CPU and a (single) GPU
- Modern computing resources provide access to many GPUs simultaneously
- It is likely that many central tasks (simulation, reconstruction) will move to Deep Learning solutions
  - We will have to train periodically large models
  - We could exploit these resources for that

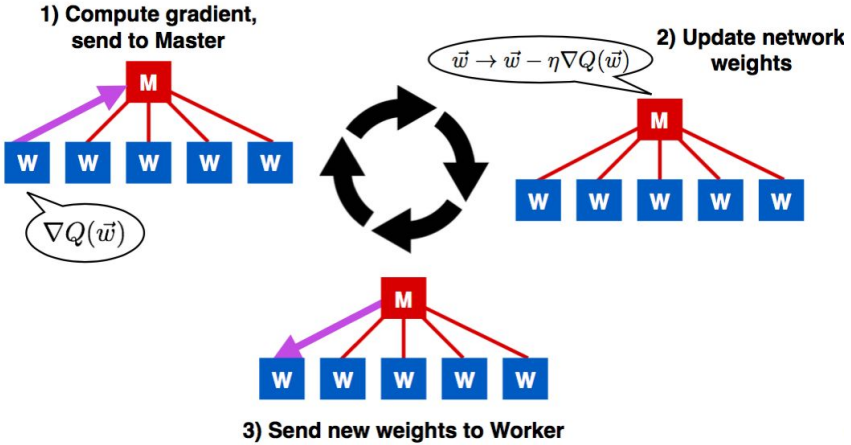
Neural network training

- Iterate:
  1) Sample a batch of data
  2) Forward it to the network to obtain the predictions
  3) Backpropagate the errors
  4) Update the weights
- (A variant) of stochastic gradient descent (SGD)
  - Adagrad, Adadelta, RMSprop, Adam...



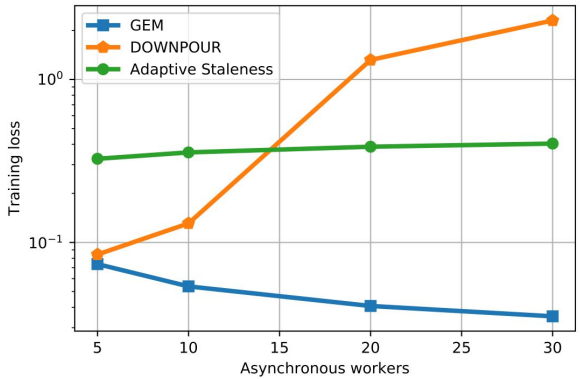
Distributed training

- Divide the training data into *N* subsets
- *N* workers **W** train a copy of the model
- Workers compute gradients and communicate with the parameter server **M**
- Parameter server handles gradients to update the central model
  - Downpour SGD
  - Elastic averaging SGD
  - Gradient Energy Matching (GEM)



Gradient energy matching

- A variant of distributed asynchronous SGD
- Core principle:
  - Make workers collectively adhere to the dynamics of a sequential SGD with momentum
  - Perform the rescaling of the gradient updates before sending them to the master node
  - Estimated energy of the asynchronous system is matched with the energy of the target proxy
- GEM ensures the stability of the collective asynchronous system
  - Assuming that the proxy converges
- Scales to a high number of workers

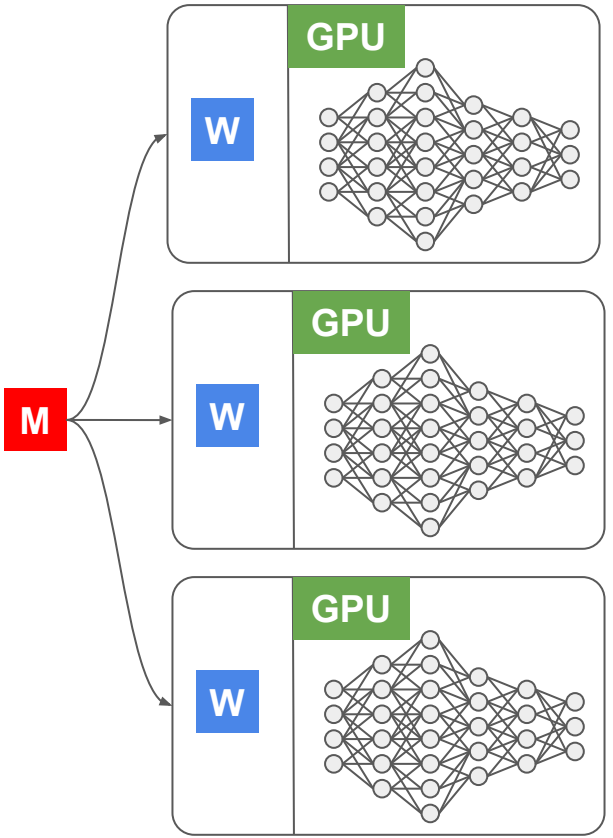


J. Hermans et al arXiv:1805.08469

Types of parallelism

- **Data parallelism**
  - Compute the gradients on several batches independently and update the model (a)synchronously
  - Applicable to large dataset

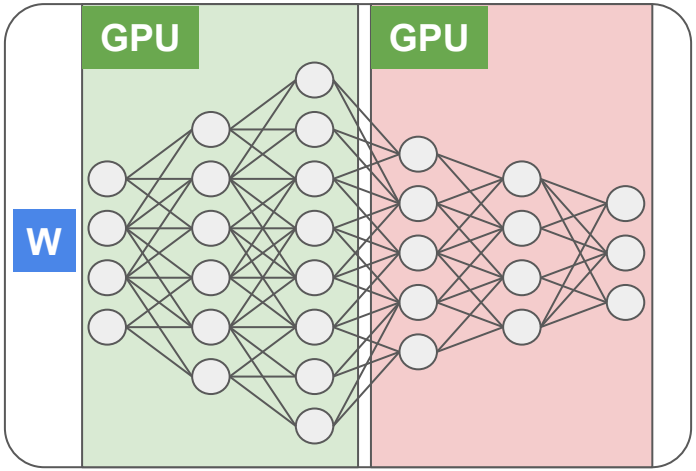
- Model parallelism
  - Compute the gradient and updates of part of the model separately in chain
  - Applicable to large model



Types of parallelism

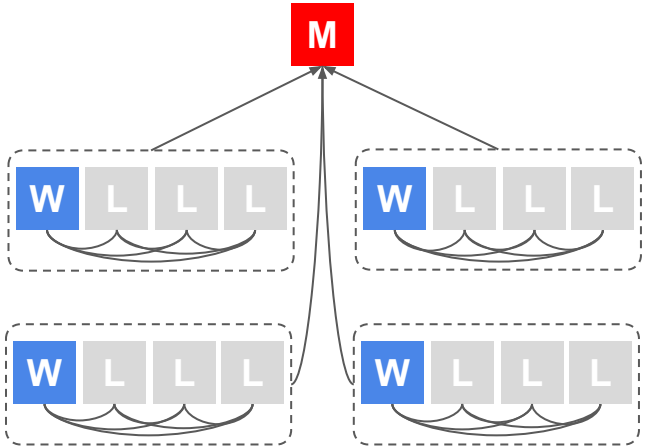
- Data parallelism
  - Compute the gradients on several batches independently and update the model (a)synchronously
  - Applicable to large dataset

- **Model parallelism**
  - Compute the gradient and updates of part of the model separately in chain
  - Applicable to large model



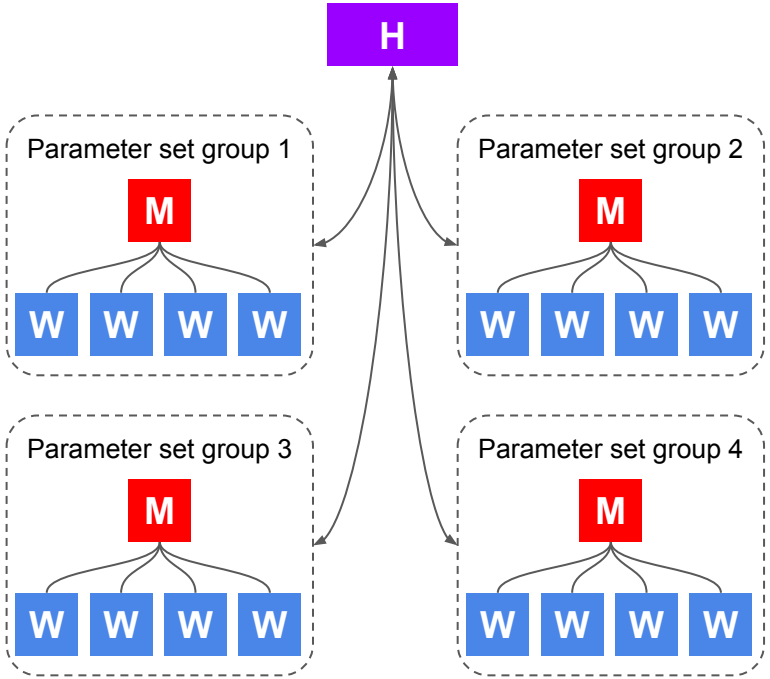
Gradient distribution

- A logical worker L is spawn over multiple MPI processes
- Workers formed using **horovod**
  - Uses NCCL for fast GPU-GPU communication
- “Allreduce” setup
  - L all work in lockstep processing a large minibatch
  - Each L computes gradients on its local shard (partition) of the minibatch
  - Send and receive gradients to/from neighbours
  - All gradients travel in the same direction on the ring
  - Once all L have received all the gradients computed for the minibatch, update the weights
  - Same copy of the model on every L after this update step



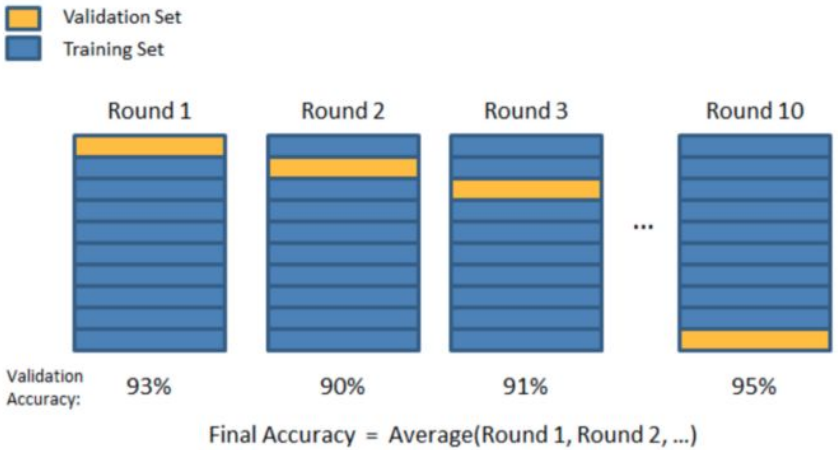
Distributed network optimization

- Various parameters of the model cannot be learned by gradient descent
  - Learning rate, batch size, number of layers, size of kernels...
- Full parameter scan is resource/time consuming
- Find a way to reach the optimum hyper-parameter set for a provided figure of merit (e.g., the loss)
- Two optimization engine integrated:
  - Bayesian optimization
  - Evolutionary algorithm



K-fold cross validation

- Estimate the performance of multiple model training over different validation part of the training dataset
- Allows to take into account variance from multiple source
  - Choice of validation set, choice of random initialization, ...
- Training on folds can proceed in parallel



NNLO library

- Neural network learning and optimization: NNLO
  - https://github.com/vlimant/NNLO/
- User-friendly library supporting multiple backends and different cluster architectures
  - Keras, PyTorch, TensorFlow
- Downpour, Elastic Averaging SGD, GEM training
- Hyperparameter optimization with bayesian optimization and evolutionary algorithm
- Checkpointing (save and restore)
- Integrated profiling and monitoring tools

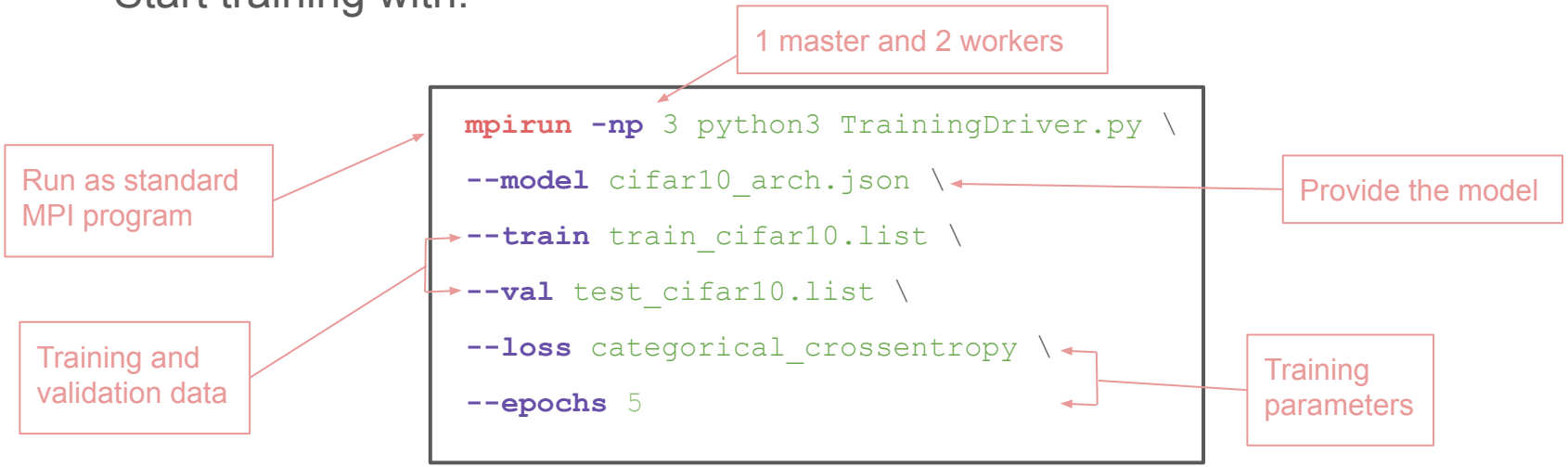
Distributing the work with MPI

- Message Passing Interface Standard (MPI) is a message passing library standard widely used for parallel programming
- Readily available on many HPC sites
- Portable code, agnostic to underlying hardware architecture
- Available in many programming languages
  - C, Fortran, Python, Java...

➜ Natural choice for NNLO

Basic training with NNLO

- User provides:
  - Model from Keras/PyTorch/TensorFlow (as JSON file, pickle, or python module)
  - Training and validation dataset
  - Choice of training algorithm and hyperparameters
- Start training with:



NNLO training interface

- Python API for training
  - Model, data and data adaptor

Full MNIST example

1
```
def get_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(1, 28, 28)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))
    return model

def get_name():
    return 'mnist'
```

Model definition

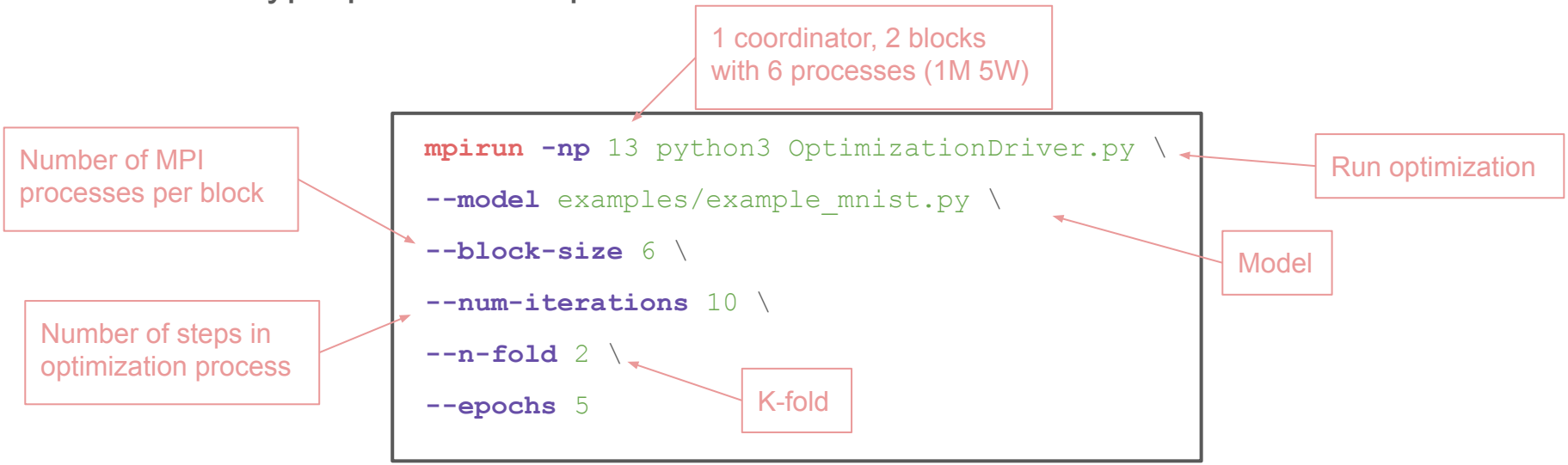
2
```
def get_train():
    all_list = glob.glob('mnist_*.h5')
    l = int(len(all_list) * 0.70)
    train_list = all_list[:l]
    return train_list

def get_val():
    all_list = glob.glob('mnist_*.h5')
    l = int(len(all_list) * 0.70)
    val_list = all_list[l:]
    return val_list

def get_features():
    return ('features', lambda x: x)

def get_labels():
    return 'labels'
```

Training and validation dataset

Data adaptor

Basic hyperparameter optimization with NNLO

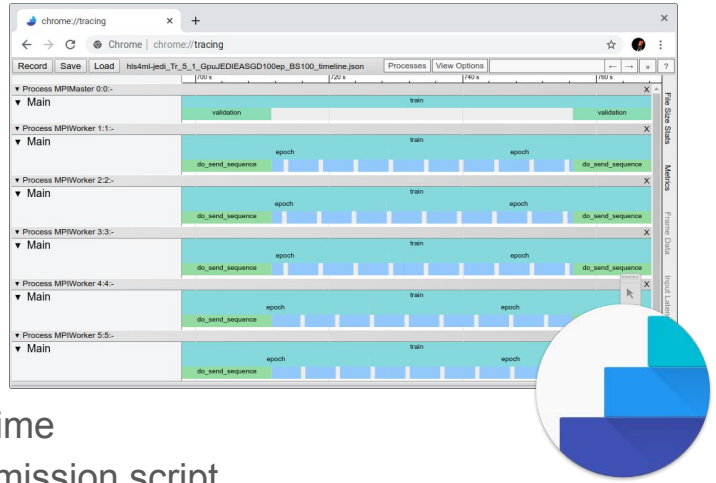
- User provides:
  - NNLO training configuration
  - Optimization parameters
- Start hyperparameter optimization with:



Useful options for HPC deployments

- Tracking resource utilization with `--monitor`
  - Tracks CPU/GPU memory usage and CPU/GPU utilization
  - Handy when tweaking a configuration for a specific cluster

- Profiling with `--timeline`
  - Traces function calls and record the execution times
  - Useful for analyzing bottlenecks



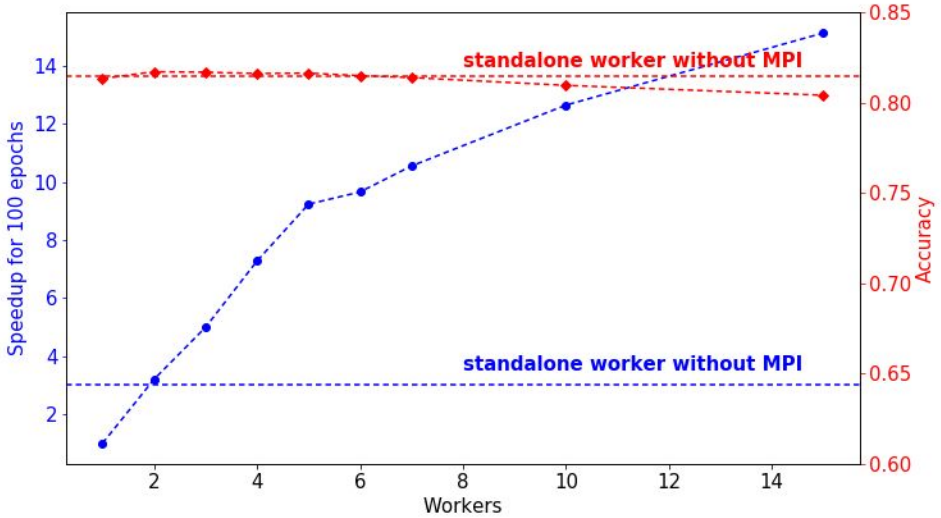
- Checkpointing with `--checkpoint`
  - Save/load functionality
  - Useful when training/optimization time > job execution time
  - Same command to checkpoint/restore ➔ same job submission script

Performance analysis

- Tested on GPU cluster at Flatiron Institute
- Computing node specs:
  - 36-core Skylake, 768GB RAM
  - 4 x Nvidia V100 with 32GB each, NVLink connected
- Software stack:
  - Open MPI 2.1
  - Python 3.7
  - CUDA 10.1
  - TensorFlow 1.14.0
  - Torch 1.1.0

Performance analysis

- JEDI-net - Graph network for particle identification (arXiv:1908.05318)
  - Computationally expensive: 33625 parameters, 116M FLOPs
  - Speedup measured w.r.t. one worker
  - Also shown performance for standalone run w/o MPI (dashed lines)



Conclusions

- NNLO library
  - https://github.com/vlimant/NNLO/
- Powerful and feature-rich library for training
  - Multiple backends: Keras, PyTorch, TensorFlow
  - Multiple optimizers: Downpour, Elastic Averaging SGD, GEM
- Hyperparameter optimization
  - Bayesian optimization and genetic algorithm
  - K-fold cross validation
- Demonstrated good scalability
- Have access to HPC and model with long training time? Contact us!