Train script

If your train script works with `torch.distributed.launch` it will continue working with `torchrun` with these differences:

1. No need to manually pass `RANK`, `WORLD_SIZE`, `MASTER_ADDR`, and `MASTER_PORT`.
2. `rdzv_backend` and `rdzv_endpoint` can be provided. For most users this will be set to `c10d` (see rendezvous). The default `rdzv_backend` creates a non-elastic rendezvous where `rdzv_endpoint` holds the master address.
3. Make sure you have a `load_checkpoint(path)` and `save_checkpoint(path)` logic in your script. When any number of workers fail we restart all the workers with the same program arguments so you will lose progress up to the most recent checkpoint (see elastic launch).
4. `use_env` flag has been removed. If you were parsing local rank by parsing the `--local_rank` option, you need to get the local rank from the environment variable `LOCAL_RANK` (e.g. `int(os.environ["LOCAL_RANK"])`).

Below is an expository example of a training script that checkpoints on each epoch, hence the worst-case progress lost on failure is one full epoch worth of training.

```
def main():
     args = parse_args(sys.argv[1:])
     state = load_checkpoint(args.checkpoint_path)
     initialize(state)

     # torch.distributed.run ensures that this will work
     # by exporting all the env vars needed to initialize the process group
     torch.distributed.init_process_group(backend=args.backend)

     for i in range(state.epoch, state.total_num_epochs)
          for batch in iter(state.dataset)
              train(batch, state.model)

          state.epoch += 1
          save_checkpoint(state)
```

For concrete examples of torchelastic-compliant train scripts, visit our examples page.