```
#r "nuget: TorchSharp"
```

Installed Packages

- TorchSharp, 0.96.7

```
#r "nuget: libtorch-cuda-11.3-linux-x64"
```

Installed Packages

- libtorch-cuda-11.3-linux-x64, 1.11.0.1

```
#r "nuget: SharpZipLib"
```

Installed Packages

- SharpZipLib, 1.3.3

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Diagnostics;

using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using ICSharpCode.SharpZipLib.Core;
using ICSharpCode.SharpZipLib.GZip;
using ICSharpCode.SharpZipLib.Tar;
```

```
public static void DecompressGZipFile(string gzipFileName, string
targetDir)
{
    byte[] buf = new byte[4096];

    using (var fs = File.OpenRead(gzipFileName))
    using (var gzipStream = new GZipInputStream(fs)) {

        string fnOut = Path.Combine(targetDir,
Path.GetFileNameWithoutExtension(gzipFileName));

        using (var fsOut = File.Create(fnOut)) {
                StreamUtils.Copy(gzipStream, fsOut, buf);
        }
    }
}
public static void ExtractTGZ(string gzArchiveName, string destFolder)
{
    var flag =
gzArchiveName.Split(Path.DirectorySeparatorChar).Last().Split('.').First()
+ ".bin";
    if (File.Exists(Path.Combine(destFolder, flag))) return;

    Console.WriteLine($"Extracting.");
    var task = Task.Run(() => {
    using (var inStream = File.OpenRead(gzArchiveName)) {
        using (var gzipStream = new GZipInputStream(inStream)) {
    #pragma warning disable CS0618 // Type or member is obsolete
            using (TarArchive tarArchive =
TarArchive.CreateInputTarArchive(gzipStream))
    #pragma warning restore CS0618 // Type or member is obsolete
                tarArchive.ExtractContents(destFolder);
            }
        }
    });

    while (!task.IsCompleted) {
        Thread.Sleep(200);
        Console.Write(".");
    }

    File.Create(Path.Combine(destFolder, flag));
    Console.WriteLine("");
    Console.WriteLine("Extraction completed.");
}
```

```
string dataPath = Path.Combine(Environment.CurrentDirectory +
"/data/minst/" , "train");
string minstPath = Environment.CurrentDirectory + "/data/minst/" ;
if (!Directory.Exists(dataPath))
{
    Directory.CreateDirectory(dataPath);
    DecompressGZipFile(Path.Combine(minstPath, "train-images-idx3-
ubyte.gz"), dataPath);
    DecompressGZipFile(Path.Combine(minstPath, "train-labels-idx1-
ubyte.gz"), dataPath);
    DecompressGZipFile(Path.Combine(minstPath, "t10k-images-idx3-
ubyte.gz"), dataPath);
    DecompressGZipFile(Path.Combine(minstPath, "t10k-labels-idx1-
ubyte.gz"), dataPath);

}
```

```
using TorchSharp;
using TorchSharp.torchvision;


using static TorchSharp.torch;

using static TorchSharp.torch.nn;
using static TorchSharp.torch.nn.functional;
```

```
var device = cuda.is_available() ? CUDA : CPU;
```

```
device
```

type	index
CUDA	-1

```
private static int _epochs = 4;
private static int _trainBatchSize = 64;
private static int _testBatchSize = 128;
```

```
public class MNISTModel : Module
{
    private Module conv1 = Conv2d(1, 32, 3);
    private Module conv2 = Conv2d(32, 64, 3);
    private Module fc1 = Linear(9216, 128);
    private Module fc2 = Linear(128, 10);

    // These don't have any parameters, so the only reason to instantiate
    // them is performance, since they will be used over and over.
    private Module pool1 = MaxPool2d(kernelSize: new long[] { 2, 2 });

    private Module relu1 = ReLU();
    private Module relu2 = ReLU();
    private Module relu3 = ReLU();

    private Module dropout1 = Dropout(0.25);
    private Module dropout2 = Dropout(0.5);

    private Module flatten = Flatten();
    private Module logsm = LogSoftmax(1);

    public MNISTModel(string name, torch.Device device = null) : base(name)
    {
        RegisterComponents();

        if (device != null && device.type == DeviceType.CUDA)
            this.to(device);
    }

    public override Tensor forward(Tensor input)
    {
        var l11 = conv1.forward(input);
        var l12 = relu1.forward(l11);

        var l21 = conv2.forward(l12);
        var l22 = relu2.forward(l21);
        var l23 = pool1.forward(l22);
        var l24 = dropout1.forward(l23);

        var x = flatten.forward(l24);

        var l31 = fc1.forward(x);
        var l32 = relu3.forward(l31);
        var l33 = dropout2.forward(l32);

        var l41 = fc2.forward(l33);

        return logsm.forward(l41);
    }
}
```

```
Console.WriteLine($"\tCreating the model...");
```

```
	Creating the model...
```

```
var model = new MNISTModel("model", device);
```

```
var normImage = transforms.Normalize(new double[] { 0.1307 }, new double[]
{ 0.3081 }, device: (Device)device);
```

```
public static int _logInterval = 100;
```

```
private static void Train(
    Module model,
    optim.Optimizer optimizer,
    Loss loss,
    Device device,
    IEnumerable<(Tensor, Tensor)> dataLoader,
    int epoch,
    long batchSize,
    long size)
{
    model.train();

    int batchId = 1;

    Console.WriteLine($"Epoch: {epoch}...");

    foreach (var (data, target) in dataLoader)
    {
        using (var d = torch.NewDisposeScope())
        {
            optimizer.zero_grad();

            var prediction = model.forward(data);
            var output = loss(prediction, target);

            output.backward();

            optimizer.step();

            if (batchId % _logInterval == 0)
            {
                Console.WriteLine($"\rTrain: epoch {epoch} [{batchId * 
batchSize} / {size}] Loss: {output.ToSingle():F4}");
            }

            batchId++;

        }
    }
}
```

```
private static void Test(
    Module model,
    Loss loss,
    Device device,
    IEnumerable<(Tensor, Tensor)> dataLoader,
    long size)
{
    model.eval();

    double testLoss = 0;
    int correct = 0;

    foreach (var (data, target) in dataLoader)
    {
        using (var d = torch.NewDisposeScope())
        {
            var prediction = model.forward(data);
            var output = loss(prediction, target);
            testLoss += output.ToSingle();

            correct += prediction.argmax(1).eq(target).sum().ToInt32();
        }
    }

    Console.WriteLine($"Size: {size}, Total: {size}");

    Console.WriteLine($"\rTest set: Average loss {(testLoss / size):F4} |
Accuracy {((double)correct / size):P2}");
}
```

```
public class BigEndianReader
{
    public BigEndianReader(BinaryReader baseReader)
    {
        mBaseReader = baseReader;
    }

    public int ReadInt32()
    {
        return BitConverter.ToInt32(ReadBigEndianBytes(4), 0);
    }

    public byte[] ReadBigEndianBytes(int count)
    {
        byte[] bytes = new byte[count];
        for (int i = count - 1; i >= 0; i--)
            bytes[i] = mBaseReader.ReadByte();

        return bytes;
    }

    public byte[] ReadBytes(int count)
    {
        return mBaseReader.ReadBytes(count);
    }

    public void Close()
    {
        mBaseReader.Close();
    }
    public Stream BaseStream {
        get { return mBaseReader.BaseStream; }
    }

    private BinaryReader mBaseReader;
}
```

```
public class MNISTReader : IEnumerable<(Tensor, Tensor)>, IDisposable
{
    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="path">Path to the folder containing the image files.
</param>
    /// <param name="prefix">The file name prefix, either 'train' or 't10k'
(the latter being the test data set).</param>
    /// <param name="batch_size">The batch size</param>
    /// <param name="shuffle">Randomly shuffle the images.</param>
    /// <param name="device">The device, i.e. CPU or GPU to place the
output tensors on.</param>
    /// <param name="transform"></param>
    public MNISTReader(string path, string prefix, int batch_size = 32,
bool shuffle = false, torch.Device device = null, ITransform transform =
null)
    {
        // The MNIST data set is small enough to fit in memory, so let's
load it there.

        BatchSize = batch_size;

        var dataPath = Path.Combine(path, prefix + "-images-idx3-ubyte");
        var labelPath = Path.Combine(path, prefix + "-labels-idx1-ubyte");

        var count = -1;
        var height = 0;
        var width = 0;

        byte[] dataBytes = null;
        byte[] labelBytes = null;

        using (var file = File.Open(dataPath, FileMode.Open,
FileAccess.Read, FileShare.Read))
        using (var rdr = new System.IO.BinaryReader(file)) {

            var reader = new BigEndianReader(rdr);
            var x = reader.ReadInt32(); // Magic number
            count = reader.ReadInt32();

            height = reader.ReadInt32();
            width = reader.ReadInt32();
            // Read all the data into memory.
            dataBytes = reader.ReadBytes(height * width * count);
        }

        using (var file = File.Open(labelPath, FileMode.Open,
FileAccess.Read, FileShare.Read))
        using (var rdr = new System.IO.BinaryReader(file)) {

            var reader = new BigEndianReader(rdr);
            var x = reader.ReadInt32(); // Magic number
            var lblcnt = reader.ReadInt32();

            if (lblcnt != count) throw new InvalidDataException("Image data
and label counts are different.");

            // Read all the data into memory.
            labelBytes = reader.ReadBytes(lblcnt);
        }

        // Set up the indices array.
        Random rnd = new Random();
        var indices = !shuffle ?
            Enumerable.Range(0, count).ToArray() :
            Enumerable.Range(0, count).OrderBy(c => rnd.Next()).ToArray();

        var imgSize = height * width;

        // Go through the data and create tensors
        for (var i = 0; i < count;) {

            var take = Math.Min(batch_size, Math.Max(0, count - i));

            if (take < 1) break;

            var dataTensor = torch.zeros(new long[] { take, imgSize},
device: device);
            var lablTensor = torch.zeros(new long[] { take }, torch.int64,
device: device);

            // Take
            for (var j = 0; j < take; j++) {
                var idx = indices[i++];
                var imgStart = idx * imgSize;

                var floats = dataBytes[imgStart..
(imgStart+imgSize)].Select(b => b/256.0f).ToArray();
                using (var inputTensor = torch.tensor(floats))
                    dataTensor.index_put_(inputTensor,
TensorIndex.Single(j));
                lablTensor[j] = torch.tensor(labelBytes[idx], torch.int64);
            }

            var batch = dataTensor.reshape(take, 1, height, width);
```

```
                if (transform != null) {
                    // Carefully dispose the original
                    using(var batch_copy = batch)
                    batch = transform.forward(batch);
                }

                data.Add(batch);
                dataTensor.Dispose();
                labels.Add(lablTensor);
            }

            Size = count;
        }

        public int Size { get; set; }

        public int BatchSize { get; private set; }

        private List<Tensor> data = new List<Tensor>();
        private List<Tensor> labels = new List<Tensor>();

        public IEnumerator<(Tensor, Tensor)> GetEnumerator()
        {
            return new MNISTEnumerator(data, labels);
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        public void Dispose()
        {
            data.ForEach(d => d.Dispose());
            labels.ForEach(d => d.Dispose());
        }

        private class MNISTEnumerator : IEnumerator<(Tensor, Tensor)>
        {
            public MNISTEnumerator(List<Tensor> data, List<Tensor> labels)
            {
                this.data = data;
                this.labels = labels;
            }

            public (Tensor, Tensor) Current {
                get {
                    if (curIdx == -1) throw new
    InvalidOperationException("Calling 'Current' before 'MoveNext()'");
                    return (data[curIdx], labels[curIdx]);
                }
            }

            object IEnumerator.Current => Current;
            public void Dispose()
            {
            }

            public bool MoveNext()
            {
                curIdx += 1;
                return curIdx < data.Count;
            }

            public void Reset()
            {
                curIdx = -1;
            }

            private int curIdx = -1;
            private List<Tensor> data = null;
            private List<Tensor> labels = null;
        }
    }
```

```
public static void TrainingLoop(string dataset, int timeout, Device device,
Module model, MNISTReader train, MNISTReader test)
{
            var optimizer = optim.Adam(model.parameters());

            var scheduler = optim.lr_scheduler.StepLR(optimizer, 1, 0.7);

            Stopwatch totalTime = new Stopwatch();
            totalTime.Start();

            for (var epoch = 1; epoch <= _epochs; epoch++)
            {

                Train(model, optimizer, nll_loss(reduction:
Reduction.Mean), device, train, epoch, train.BatchSize, train.Size);
                Test(model, nll_loss(reduction: nn.Reduction.Sum), device,
test, test.Size);

                Console.WriteLine($"End-of-epoch memory use:
{GC.GetTotalMemory(false)}");

                if (totalTime.Elapsed.TotalSeconds > timeout) break;
            }

            totalTime.Stop();
            Console.WriteLine($"Elapsed time:
{totalTime.Elapsed.TotalSeconds:F1} s.");

            Console.WriteLine("Saving model to '{0}'", dataset +
".model.bin");
            model.save(dataset + ".model.bin");
}
```

```
using (MNISTReader train = new MNISTReader(dataPath, "train",
_trainBatchSize, device: device, shuffle: true, transform: normImage),
test = new MNISTReader(dataPath, "t10k", _testBatchSize, device: device,
transform: normImage))
{

    TrainingLoop("mnist", 3000, device, model, train, test);
}
```

```
Epoch: 1...
Train: epoch 1 [6400 / 60000] Loss: 0.7495
Train: epoch 1 [12800 / 60000] Loss: 0.4441
Train: epoch 1 [19200 / 60000] Loss: 0.5683
Train: epoch 1 [25600 / 60000] Loss: 0.5474
Train: epoch 1 [32000 / 60000] Loss: 0.2864
Train: epoch 1 [38400 / 60000] Loss: 0.4358
Train: epoch 1 [44800 / 60000] Loss: 0.3547
Train: epoch 1 [51200 / 60000] Loss: 0.3173
Train: epoch 1 [57600 / 60000] Loss: 0.2953
Size: 10000, Total: 10000
Test set: Average loss 0.3069 | Accuracy 88.48%
End-of-epoch memory use: 680042136
Epoch: 2...
Train: epoch 2 [6400 / 60000] Loss: 0.5452
Train: epoch 2 [12800 / 60000] Loss: 0.2801
Train: epoch 2 [19200 / 60000] Loss: 0.3832
Train: epoch 2 [25600 / 60000] Loss: 0.3985
Train: epoch 2 [32000 / 60000] Loss: 0.3380
Train: epoch 2 [38400 / 60000] Loss: 0.2657
Train: epoch 2 [44800 / 60000] Loss: 0.2054
Train: epoch 2 [51200 / 60000] Loss: 0.3458
Train: epoch 2 [57600 / 60000] Loss: 0.2508
Size: 10000, Total: 10000
Test set: Average loss 0.2542 | Accuracy 90.78%
End-of-epoch memory use: 700045024
Epoch: 3...
Train: epoch 3 [6400 / 60000] Loss: 0.3732
Train: epoch 3 [12800 / 60000] Loss: 0.4115
Train: epoch 3 [19200 / 60000] Loss: 0.3520
Train: epoch 3 [25600 / 60000] Loss: 0.3260
Train: epoch 3 [32000 / 60000] Loss: 0.1877
Train: epoch 3 [38400 / 60000] Loss: 0.2346
Train: epoch 3 [44800 / 60000] Loss: 0.2435
Train: epoch 3 [51200 / 60000] Loss: 0.1471
Train: epoch 3 [57600 / 60000] Loss: 0.1659
```

```
Size: 10000, Total: 10000
Test set: Average loss 0.2335 | Accuracy 91.59%
End-of-epoch memory use: 720057296
Epoch: 4...
Train: epoch 4 [6400 / 60000] Loss: 0.3384
Train: epoch 4 [12800 / 60000] Loss: 0.3029
Train: epoch 4 [19200 / 60000] Loss: 0.2921
Train: epoch 4 [25600 / 60000] Loss: 0.3197
Train: epoch 4 [32000 / 60000] Loss: 0.1415
Train: epoch 4 [38400 / 60000] Loss: 0.1327
Train: epoch 4 [44800 / 60000] Loss: 0.2397
Train: epoch 4 [51200 / 60000] Loss: 0.1498
Train: epoch 4 [57600 / 60000] Loss: 0.1642
Size: 10000, Total: 10000
Test set: Average loss 0.2422 | Accuracy 91.48%
End-of-epoch memory use: 740123272
Elapsed time: 11.1 s.
Saving model to 'mnist.model.bin'
```