```
#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

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

using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Text.RegularExpressions;
```

```
string newsPath = Environment.CurrentDirectory + "/data/ag_news/" ;

long emsize = 200;

long batch_size = 128;
long eval_batch_size = 128;

int epochs = 15;
```

```
using TorchSharp;
using static TorchSharp.torch;

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

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

```
device
```

type	index
CUDA	-1

```
public static class Utils
{
    public static Func<string, IEnumerable<string>> get_tokenizer(string
name)
    {
        if (name == "basic_english") return BasicEnglish;
        throw new NotImplementedException($"The '{name}' text tokenizer is
not implemented.");
    }

    private static string[] _patterns = new string []{
         "\'",
         "\"",
         "\\.",
         "<br \\/>",
         ",",
         "\\(",
         "\\)",
         "\\!",
         "\\?",
         "\\;",
         "\\:",
         "\\\\",
         "\\s+",
    };
    private static string[] _replacements = new string[] {
         " \\'  ",
         "",
         " . ",
         " ",
         " , ",
         " ( ",
         " ) ",
         " ! ",
         " ? ",
         " ",
         " ",
         " ",
         " "
    };

    private static IEnumerable<string> BasicEnglish(string input)
    {
        if (_patterns.Length != _replacements.Length)
```

```
            throw new InvalidProgramException("internal error: patterns and
replacements are not the same length");

        input = input.Trim().ToLowerInvariant();

        for (var i = 0; i < _patterns.Length; ++i) {
            input = Regex.Replace(input, _patterns[i], _replacements[i]);
        }
        return input.Split(' ');
    }
}
```

```
public class Counter<T> : IEnumerable<KeyValuePair<T, int>>
{
        private Dictionary<T, int> _dict = new Dictionary<T, int>();

        public void update(T key)
        {
            if (_dict.TryGetValue(key, out int count)) {
                _dict[key] = count + 1;
            } else {
                _dict[key] = 1;
            }
        }
        public void update(IEnumerable<T> keys)
        {
            foreach (T key in keys) {
                update(key);
            }
        }
        public int this[T key] { get => _dict[key]; }

        public IEnumerator<KeyValuePair<T, int>> GetEnumerator()
        {
            return _dict.GetEnumerator();
        }

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


public class Vocab
{
        public Vocab(Counter<string> counter, int? maxSize = null, int
minFreq = 1, string[] specials = null, Func<torch.Tensor, torch.Tensor>
unkInit = null, bool specialsFirst = true)
        {
            if (specials == null) specials = new string[] { "<unk>", "
```

```
<pad>" };
            if (unkInit == null) unkInit = (t => init.zeros_(t.clone()));
            if (specialsFirst) {
                foreach (var sp in specials) {
                    _dict.Add(sp, _last++);
                }
            }
            foreach (var kv in counter.Where(kv => kv.Value >= minFreq)) {
                if (!specials.Contains(kv.Key)) {
                    _dict.Add(kv.Key, _last++);
                }
                if (_last > (maxSize ?? int.MaxValue))
                    break;
            }
            if (!specialsFirst) {
                foreach (var sp in specials) {
                    _dict.Add(sp, _last++);
                }
            }
        }

        public int this[string key] { get => _dict.TryGetValue(key, out int
value) ? value : _dict["<unk>"]; }

        public int Count => _dict.Count;

        public void Add(string key, int value)
        {
            _dict.Add(key, value);
        }

        public void Add(KeyValuePair<string, int> item)
        {
            Add(item.Key, item.Value);
        }

        public bool TryGetValue(string key, [MaybeNullWhen(false)] out int
value)
        {
            return _dict.TryGetValue(key, out value);
        }

        private Dictionary<string, int> _dict = new Dictionary<string, int>
();
        private int _last = 0;
}
```

```
public class AG_NEWSReader : IDisposable
{
        public static AG_NEWSReader AG_NEWS(string split, Device device,
string root = ".data")
```

```
        {
            var dataPath = Path.Combine(root, $"{split}.csv");
            Console.WriteLine(dataPath);
            return new AG_NEWSReader(dataPath, device);
        }

        private AG_NEWSReader(string path, Device device)
        {
            _path = path;
            _device = device;
        }

        private string _path;
        private Device _device;

        public IEnumerable<(int, string)> Enumerate()
        {
            return File.ReadLines(_path).Select(line => ParseLine(line));
        }

        public IEnumerable<(Tensor, Tensor, Tensor)>
GetBatches(Func<string, IEnumerable<string>> tokenizer, Vocab vocab, long
batch_size)
        {

            var inputs = new List<(int, string)>();

            if (_data == null) {

                _data = new List<(Tensor, Tensor, Tensor)>();

                var counter = 0;
                var lines = Enumerate().ToList();
                var left = lines.Count;

                foreach (var line in lines) {

                    inputs.Add(line);
                    left -= 1;

                    if (++counter == batch_size || left == 0) {
                        _data.Add(Batchifier(inputs, tokenizer, vocab));
                        inputs.Clear();
                        counter = 0;
                    }
                }
            }

            return _data;
        }

        private List<(Tensor, Tensor, Tensor)> _data;
        private bool disposedValue;
```

```
        private (Tensor, Tensor, Tensor) Batchifier(IEnumerable<(int,
string)> input, Func<string, IEnumerable<string>> tokenizer, Vocab vocab)
        {
            var label_list = new List<long>();
            var text_list = new List<Tensor>();
            var offsets = new List<long>();
            offsets.Add(0);

            long last = 0;

            foreach (var (label, text) in input) {
                label_list.Add(label);
                var processed_text = torch.tensor(tokenizer(text).Select(t
=> (long)vocab[t]).ToArray(),dtype:torch.int64);
                text_list.Add(processed_text);
                last += processed_text.size(0);
                offsets.Add(last);
            }

            var labels = torch.tensor(label_list.ToArray(), dtype:
torch.int64).to(_device);
            var texts = torch.cat(text_list.ToArray(), 0).to(_device);
            var offs =
torch.tensor(offsets.Take(label_list.Count).ToArray(),
dtype:torch.int64).to(_device);

            return (labels, texts, offs);
        }

        public (int, string) ParseLine(string line)
        {
            int label = 0;
            string text = "";

            int firstComma = line.IndexOf("\",\"");
            label = int.Parse(line.Substring(1, firstComma - 1));
            text = line.Substring(firstComma + 2, line.Length - firstComma
- 2);
            int secondComma = text.IndexOf("\",\"");
            text = text.Substring(secondComma + 2, text.Length -
secondComma - 2);
            int thirdComma = text.IndexOf("\",\"");

            text = text.Substring(thirdComma + 2, text.Length - thirdComma
- 3);

            return (label-1, text);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue) {
                if (disposing && _data != null) {
                    foreach (var (l, t, o) in _data) {
```

```
                        l.Dispose();
                        t.Dispose();
                        o.Dispose();
                    }
                }

                disposedValue = true;
            }
        }

        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
}
```

```
var reader = AG_NEWSReader.AG_NEWS("train", (Device)device, newsPath);
```

```
/home/lokinfey/Dev/ML/dotnet/torch/data/ag_news/train.csv
```

```
var dataloader = reader.Enumerate();

var tokenizer = Utils.get_tokenizer("basic_english");

var counter = new Counter<string>();

foreach (var (label, text) in dataloader)
{
    counter.update(tokenizer(text));
}
```

```
var vocab = new Vocab(counter);
```

```
public class TextClassificationModel : Module
{
        private TorchSharp.Modules.EmbeddingBag embedding;
        private TorchSharp.Modules.Linear fc;

        public TextClassificationModel(long vocab_size, long embed_dim,
long num_class) : base("TextClassification")
        {
```

```
            embedding = EmbeddingBag(vocab_size, embed_dim, sparse: false);
            fc = Linear(embed_dim, num_class);
            InitWeights();

            RegisterComponents();
        }

        private void InitWeights()
        {
            var initrange = 0.5;

            init.uniform_(embedding.weight, -initrange, initrange);
            init.uniform_(fc.weight, -initrange, initrange);
            init.zeros_(fc.bias);
        }

        public override Tensor forward(Tensor t)
        {
            throw new NotImplementedException();
        }

        public override Tensor forward(Tensor input, Tensor offsets)
        {
            var t = embedding.forward(input, offsets);
            return fc.forward(t);
        }

        public new TextClassificationModel to(Device device)
        {
            base.to(device);
            return this;
        }
}
```

```
static void train(int epoch, IEnumerable<(Tensor, Tensor, Tensor)>
train_data, TextClassificationModel model, Loss criterion,
torch.optim.Optimizer optimizer)
{
    model.train();

    double total_acc = 0.0;
    long total_count = 0;
    long log_interval = 250;

    var batch = 0;

    var batch_count = train_data.Count();

    using (var d = torch.NewDisposeScope())
    {
        foreach (var (labels, texts, offsets) in train_data)
```

```
        {

            optimizer.zero_grad();

            using (var predicted_labels = model.forward(texts, offsets))
            {

                var loss = criterion(predicted_labels, labels);
                loss.backward();
                torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5);
                optimizer.step();

                total_acc += (predicted_labels.argmax(1) ==
labels).sum().to(torch.CPU).item<long>();
                total_count += labels.size(0);
            }

            if (batch % log_interval == 0 && batch > 0)
            {
                var accuracy = total_acc / total_count;
                Console.WriteLine($"epoch: {epoch} | batch: {batch} /
{batch_count} | accuracy: {accuracy:0.00}");
            }
            batch += 1;
        }
    }
}
```

```
static double evaluate(IEnumerable<(Tensor, Tensor, Tensor)> test_data,
TextClassificationModel model, Loss criterion)
{
            model.eval();

            double total_acc = 0.0;
            long total_count = 0;

            using (var d = torch.NewDisposeScope())
            {
                foreach (var (labels, texts, offsets) in test_data)
                {

                    using (var predicted_labels = model.forward(texts,
offsets))
                    {
                        var loss = criterion(predicted_labels, labels);

                        total_acc += (predicted_labels.argmax(1) ==
labels).sum().to(torch.CPU).item<long>();
                        total_count += labels.size(0);
                    }
                }
```

```
                return total_acc / total_count;
            }
}
```

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

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

```
var model = new TextClassificationModel(vocab.Count, emsize,
4).to((Device)device);
```

```
var loss = cross_entropy_loss();
var lr = 5.0;
var optimizer = torch.optim.SGD(model.parameters(), lr);
var scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, 0.2,
last_epoch: 5);
```

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

int timeout = 3000;
```

```
foreach (var epoch in Enumerable.Range(1, epochs))
{

    var sw = new Stopwatch();
    sw.Start();

    train(epoch, reader.GetBatches(tokenizer, vocab, batch_size), model,
loss, optimizer);

    sw.Stop();

    Console.WriteLine($"\nEnd of epoch: {epoch} | lr:
{optimizer.ParamGroups.First().LearningRate:0.0000} | time:
{sw.Elapsed.TotalSeconds:0.0}s\n");
    scheduler.step();
```

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

totalTime.Stop();
```

```
epoch: 1 | batch: 250 / 938 | accuracy: 0.49
epoch: 1 | batch: 500 / 938 | accuracy: 0.58
epoch: 1 | batch: 750 / 938 | accuracy: 0.63

End of epoch: 1 | lr: 1.0000 | time: 5.5s

epoch: 2 | batch: 250 / 938 | accuracy: 0.79
epoch: 2 | batch: 500 / 938 | accuracy: 0.80
epoch: 2 | batch: 750 / 938 | accuracy: 0.80

End of epoch: 2 | lr: 0.2000 | time: 2.3s

epoch: 3 | batch: 250 / 938 | accuracy: 0.81
epoch: 3 | batch: 500 / 938 | accuracy: 0.81
epoch: 3 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 3 | lr: 0.0400 | time: 2.3s

epoch: 4 | batch: 250 / 938 | accuracy: 0.81
epoch: 4 | batch: 500 / 938 | accuracy: 0.81
epoch: 4 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 4 | lr: 0.0080 | time: 2.3s

epoch: 5 | batch: 250 / 938 | accuracy: 0.81
epoch: 5 | batch: 500 / 938 | accuracy: 0.81
epoch: 5 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 5 | lr: 0.0016 | time: 2.3s

epoch: 6 | batch: 250 / 938 | accuracy: 0.81
epoch: 6 | batch: 500 / 938 | accuracy: 0.81
epoch: 6 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 6 | lr: 0.0003 | time: 2.3s

epoch: 7 | batch: 250 / 938 | accuracy: 0.81
epoch: 7 | batch: 500 / 938 | accuracy: 0.81
epoch: 7 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 7 | lr: 0.0001 | time: 2.3s

epoch: 8 | batch: 250 / 938 | accuracy: 0.81
epoch: 8 | batch: 500 / 938 | accuracy: 0.81
epoch: 8 | batch: 750 / 938 | accuracy: 0.82
```

```
End of epoch: 8 | lr: 0.0000 | time: 2.3s

epoch: 9 | batch: 250 / 938 | accuracy: 0.81
epoch: 9 | batch: 500 / 938 | accuracy: 0.81
epoch: 9 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 9 | lr: 0.0000 | time: 2.3s

epoch: 10 | batch: 250 / 938 | accuracy: 0.81
epoch: 10 | batch: 500 / 938 | accuracy: 0.81
epoch: 10 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 10 | lr: 0.0000 | time: 2.3s

epoch: 11 | batch: 250 / 938 | accuracy: 0.81
epoch: 11 | batch: 500 / 938 | accuracy: 0.81
epoch: 11 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 11 | lr: 0.0000 | time: 2.3s

epoch: 12 | batch: 250 / 938 | accuracy: 0.81
epoch: 12 | batch: 500 / 938 | accuracy: 0.81
epoch: 12 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 12 | lr: 0.0000 | time: 2.3s

epoch: 13 | batch: 250 / 938 | accuracy: 0.81
epoch: 13 | batch: 500 / 938 | accuracy: 0.81
epoch: 13 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 13 | lr: 0.0000 | time: 2.3s

epoch: 14 | batch: 250 / 938 | accuracy: 0.81
epoch: 14 | batch: 500 / 938 | accuracy: 0.81
epoch: 14 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 14 | lr: 0.0000 | time: 2.3s

epoch: 15 | batch: 250 / 938 | accuracy: 0.81
epoch: 15 | batch: 500 / 938 | accuracy: 0.81
epoch: 15 | batch: 750 / 938 | accuracy: 0.82

End of epoch: 15 | lr: 0.0000 | time: 2.3s
```

```
using (var test_reader = AG_NEWSReader.AG_NEWS("test", (Device)device,
newsPath))
{

    var sw = new Stopwatch();
    sw.Start();
```

```
    var accuracy = evaluate(test_reader.GetBatches(tokenizer, vocab,
eval_batch_size), model, loss);

    sw.Stop();

    Console.WriteLine($"\nEnd of training: test accuracy: {accuracy:0.00} |
eval time: {sw.Elapsed.TotalSeconds:0.0}s\n");
    scheduler.step();
}
```

```
/home/lokinfey/Dev/ML/dotnet/torch/data/ag_news/test.csv

End of training: test accuracy: 0.81 | eval time: 0.2s
```