Assignment 1 Part 2 - Phoneme Recognition

Introduction to Deep Learning
Emeritus & Carnegie Mellon University

1 Overview

In this assignment, we'll be tackling a real-world application of DL using the official PyTorch.

Objective: Our task is from the domain of speech recognition. Given audio of human speech, our network will attempt to classify the "phoneme" being spoken at each 25ms frame of audio.

Note: We recommend you start this assignment early. While understanding the concepts and writing the code may not take more than a few hours, total training time may take > 4 (although you can be AFK for much of it).

That said, to save you some time, we provide advice and guidelines on selecting architectures/hyperparams. You hopefully won't need to train more than 2 models.

In practice, people can end up training dozens to thousands of models in order to find good architectures and hyperparams. Even with dozens of GPUs, this can take weeks. But almost all of those models lead to dead-ends. Our suggestions will still give you a good taste of what choices matter without wasting your time

Before we dive in, let's cover a few important concepts.

Note: You may have noticed that deep learning involves a *lot* of reading and misc domain knowledge. This never changes, even after you get more comfortable with DL. If anything, you read MORE the deeper you go. So you'll have to get used to powering through readings; you can do it!

1.1 Background: Prepping audio data for phoneme classification

We'll first cover how audio data is typically prepped for speech processing tasks.

Raw audio data (i.e. a 'speech waveform') is a sequence of numbers that represent the amplitude of the sound wave at each time step.

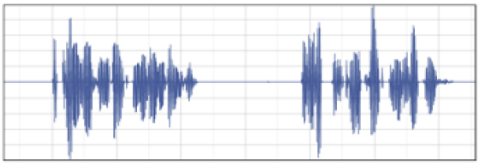

Figure 1: Typical example of raw audio data. X axis is time step and Y axis is amplitude.

However, for various reasons, this raw 1D signal generally isn't great for speech processing applications. So we convert this data to a **mel-frequency spectrogram**.

1.1.1 Spectrograms

A **spectrogram** represents a signal across time decomposed into amplitudes of individual frequencies.

One reason we prefer spectrograms over raw audio is that raw audio only tells us the total amplitude at each time. But the amplitude of each individual frequency may be super informative. So we convert to spectrograms to have that info readily available.

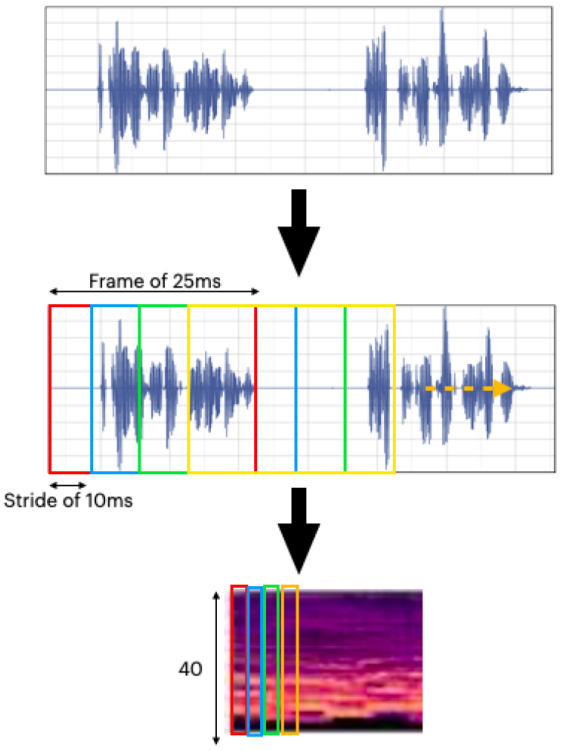

Figure 2: Converting raw audio to a spectrogram.

Without getting into too much depth, this is the general process for how this conversion happens:

1. We split the raw data into overlapping "frames" of pre-defined duration.
    - We define each frame size (here 25ms) and the 'stride' (here 10ms).
    - Starting from the beginning of the signal, we grab a frame covering 25ms, then move forward 10ms after the beginning of this frame and grab another frame covering 25ms.
    - Notice each frame slightly overlaps with its neighbors.
2. We perform a Fourier transform on each frame.
3. We apply filters to extract the amplitude of each frequency band (typically 40 total filters for 40 total frequency bands).

So a 10 second audio clip would get converted into a matrix shaped `(1000, 40)`. We get 1000 because a stride of 10ms means we'll get 100 frames per second, and there are 10 seconds. `100 * 10 = 1000`. We get 40 because of the 40 frequency bands extracted.

If you're interested in the details of the process, you can read about it here.

1.1.2 Mel-Frequency

To improve suitability for speech even further, we can adjust how we define each filter according to the **Mel-scale**. This scale adjusts the relative strength of each frequency band to mimic the logarithmic way that humans perceive sound. Like your own hearing, this scale will emphasize lower frequencies and de-emphasize higher ones.

Now we have mel-frequency spectrograms!

Note: We've already converted the data to mel-spectrograms for you. But we still cover all this to make sure you understand, end-to-end, how this pipeline works.

1.2 Background: Phonemes

Now let's introduce **phonemes**, which are the labels of our problem.

Phonemes are indivisible units of human speech. Just like how English writing can be viewed as a sequence of letters, speech can be viewed as a sequence of phonemes.

For our problem, we'll be classifying each frame as being one of 71 possible sounds and phonemes:

```
['SIL', 'SPN', 'AA0', 'AA1', 'AA2', 'AE0', 'AE1', 'AE2', 'AH0',
'AH1', 'AH2', 'AO0', 'AO1', 'AO2', 'AW0', 'AW1', 'AW2', 'AY0', 'AY1',
'AY2', 'B', 'CH', 'D', 'DH', 'EH0', 'EH1', 'EH2', 'ER0', 'ER1',
'ER2', 'EY0', 'EY1', 'EY2', 'F', 'G', 'HH', 'IH0', 'IH1', 'IH2',
'IY0', 'IY1', 'IY2', 'JH', 'K', 'L', 'M', 'N', 'NG', 'OW0', 'OW1',
'OW2', 'OY0', 'OY1', 'OY2', 'P', 'R', 'S', 'SH', 'T', 'TH', 'UH0',
'UH1', 'UH2', 'UW0', 'UW1', 'UW2', 'V', 'W', 'Y', 'Z', 'ZH']
```

Some notes:

- `'SIL'` represents silence (no phonemes being spoken)
- `'SPN'` represents an unknown sound (like a cough)
- Phonemes with a number after them (i.e. '`OW2`') indicate similar but slightly different variations of a phoneme.
- There isn't necessarily one gold-standard set of phonemes for a single language. Different datasets and languages you see in the future may use different phonemes.

Here are some examples of spoken words broken down into phonemes:

```
Erupt: IH0 R AH1 P T
Existant: EH0 G Z IH1 S T AH0 N T
Twitter: T W IH1 T AH R
```

If you want any additional info, these two Wikipedia articles are excellent:

1. Phoneme
2. English Phonology

2 Classifying Phonemes

Almost there!

Now that we understand our data (mel-spectrograms) and labels (phonemes), we can more precisely state our goal.

Objective: Identify the phoneme being spoken for every frame in the dataset.

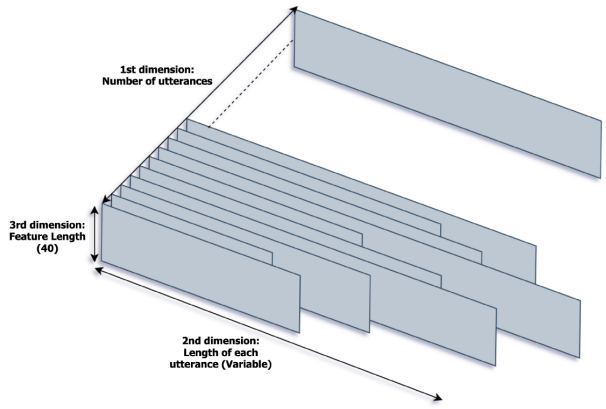

Figure 3: Dataset containing mel-spectrograms.

Above is an illustration of the dataset.

Notice:

- Each horizontal rectangle is a single audio clip (converted into mel-spectrograms).
- Every audio clip has a `num_freqs` dimension of size 40 (the 'height' of the rectangle)
- Most importantly, notice that the length (`num_frames`) of each audio clip differs. Some utterances are long, and some are short.

In summary, each audio clip will be of shape `(num_frames, num_freqs=40)`, with variable `num_frames`.

But we're not trying to classify audio clips - we're trying to classify each frame in each audio clip. So a batch of observations given to the NN will be shaped `(batch_size, num_freqs=40)`.
So there'll be one label (an integer index of the correct phoneme) for every frame of every recording in our dataset. So if there were 5 recordings in our dataset, and each contained 20 frames, there'd be 100 frames and 100 labels.

So predicting the phoneme for each frame seems straightforward, right? Just feed in a single frame and get a prediction for that frame. Unfortunately it's not that simple... (see next page)

2.1 Adding Context

Each frame represents just 25ms of speech. That's very short; phonemes generally take around 50-400ms, with some even taking up to 800ms.

So inputting and classifying just a single frame won't work. **But what if we provide some adjacent frames as context**?

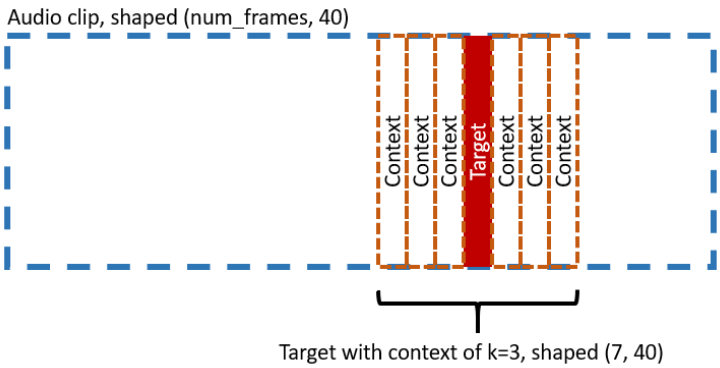

Figure 4: Adding context

To add context, we grab `k` frames from each side and attach them to our target frame. The shape of our target along with the attached context will be `(2*k+1, 40)`, where k is the context.

We'd still be predicting a single phoneme (the target frame's label), there's just some context around it now.

Also note that we've already completed this context-grabbing code for you - this description is just so you understand what's going on.

So with this context, how many ms do we cover now? Is it enough to capture a phoneme?

We can calculate how much audio we cover using this formula:

```
num_ms(k) = stride * (2 * k + 1) + frame_width
```

So if `k=3`, and given a stride of 10ms and frame width of 25ms, our observation now covers 95ms, which is a bit short.

See the next page for a plot of this function.

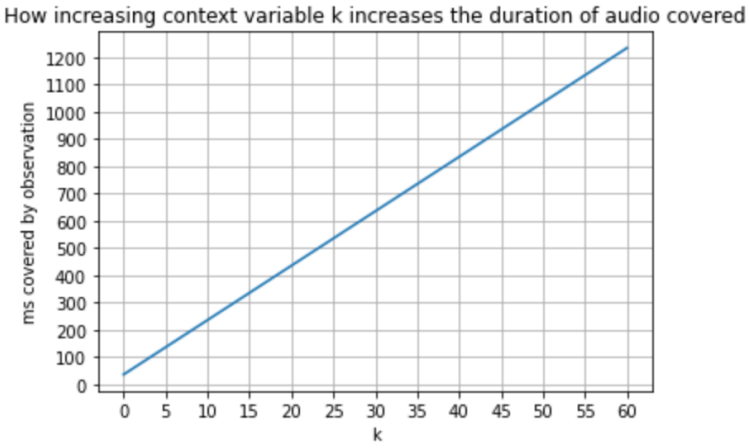

Figure 5: Increasing context helps!

Start by covering around 700-800ms and increase if you need a performance boost.

Increasing context further will likely be beneficial but may (eventually) hurt your training:

- Benefits of increasing context
    1. Increasing certainty your input covers the entire target phoneme
    2. Even if you overshoot and end up including some neighboring phonemes, those neighbors may be correlated with the target phoneme (e.g. certain vowel sounds tend to be adjacent to certain consonant sounds)
- (Eventual) Detriments
    1. Slower training, due to larger inputs, larger network needed to process them, and more parameters that need to be tuned.
    2. If your input captures too many phonemes, the network will have to learn which to pay attention to and which to ignore. Likely the further away a phoneme is to your target, the less correlated it will be, meaning decreasing returns.

You can always start training and see how long `tqdm` (the progress bar we have set up for you) estimates an epoch will take. If it's too long, you can decrease `k` to speed things up.

3 Assignment

We've provided you a training dataset, validation dataset, and testing dataset in the `data/` folder. Notice that the test dataset is the only one without corresponding labels.

Your goal is to design and train a model that gets the best accuracy on a test dataset that we've hidden the labels to.

Here's an overview of the assignment:

1. Select an architecture, optimizer, and hyperparams for your model
2. Train for some number of epochs, validate once per epoch
3. After an initial number of epochs trained, decide if you want to continue training
    - Based on if there's still clear potential for improvement or if you're adequately satisfied with your validation performance
4. Given your test dataset, generating a list of predictions (one int per frame)
5. Exporting this list to a `.csv` file and submitting it for grading.

You may make as many test submissions as you'd like. Your grade will be determined by the highest score you achieve, even if you submit a lower score later.

In order to encourage friendly competition, a small part of your grade will be partially determined by the performance of your peers. You can see how they're doing on a public leaderboard. But rest assured - implementing the baseline described in the appendix and perhaps iterating on it once or twice will be enough to do well.

Last caveat: for this assignment, you are only allowed to use a feed-forward network that you implement yourself.

This means:

- No convolutional, pooling, recurrent, transformer, or geometric layers
- No attention mechanisms.
- No using pretrained DL models/ML algorithms
- You're encouraged to use these layers: `nn.Linear`, `nn.BatchNorm1d`, `nn.Dropout`, and any activation function you wish (except for the attention one). It's possible other layer types are acceptable, but unlikely. Focus on what we've covered in this write-up and so far in the course.
- The rest is fair game (e.g. batch size, optimizers, dropout percentage, weight init, class weighting in loss function, data augmentation, etc)

We limit your options to have you genuinely learn the fundamentals of DL. There's an ocean of obscure architectures/components, and diving in immediately generally leads to only surface-level understandings and a rip tide of bugs (ocean metaphor ends here).

Now you can begin coding. Open the provided Jupyter notebook and follow the instructions.

Appendices

Appendices A-D have very good info (distilled over lots of experience and research), but if you're short on time, skip to appendix E for our recommended pipeline.

A Layer Widths and Model Depth

Some rules of thumb:

- To start, make your network as wide/deep as possible given your memory/time constraints.
  - To check for this, train for one iter on GPU. After backprop, your memory consumption will be largest: your weights, gradients, and data are stored on GPU. If it doesn't crash, it fits. You can also verify this by running `!nvidia-smi` and seeing how much memory is used.
    - Then try running for one epoch. `tqdm` (the progress bar) will estimate how long it will take; if it's too long, you can stop the execution and reduce model size or increase batch size, and see if that helps.
  - Ideally, you'll overfit. This is surprisingly a good thing; it means your network has enough capacity to memorize data, which in modern DL theory, suggests potential for good learning. You can add more regularization or decrease the size of your network in response.
  - If you underfit, consider changing architecture.
  - If your performance is near randomness, check for bugs.
- Which to increase, depth or width? Depth, but with some caveats.
  - Deeper networks can be more expressive with fewer parameters.
    - As discussed in lecture, a 2-layer neural network (linear, activation, linear) is a universal approximator, although it may need to be exponentially wider than the number of input features. In many modern DL problems, this is infeasible.
    - To avoid this, we introduce depth, which significantly reduces the number of parameters needed to approximate the same function.
  - Caveat 1: Eventually, deep-enough networks can lead to unstable learning due to "vanishing/exploding gradients".
    - This is unlikely to matter that much for this assignment, but it will in the next.
    - There are ways to get around this problem, which you'll use eventually (but not now).
  - Caveat 2: If you sacrifice too much width in order to get depth, you can lose important richness of information at each layer in the model.
  - **Solution**: Make sure each layer is just "wide enough" and prioritize depth.
    - The definition of "wide enough" varies; see next bullet point.
- Generally, it's good to have bigger layers earlier in the network, then gradually decrease layer size as you go on.
  - Your first layer could have `out_features` $\geq \frac{1}{2} *$ `in_features`.
  - Your second-to-last layer (layer before the output layer) should have `out_features` $\geq$ the output layer's `out_features`.
  - If you restrict layer size too early or go smaller than the number of input dimensions, you force the information through a bottleneck, which may cause useful info to be lost.
    - Bottlenecks aren't always bad (your own brain does this in the visual cortex), but doing it carelessly or too early in the network will likely hurt.

B Batch Normalization

Batch Normalization ("BatchNorm") is a successful and popular technique for improving the speed and quality of learning in NNs. Although its exact reason for helping is still heavily debated, it's suspected that it makes the values at each layer stabler, making the learning problem easier.

B.1 How it works

To do this, it essentially normalizes/whitens the values *between* layers. Specifically, a BN layer aims to linearly transform the output of the previous layer s.t. across the entire dataset, each neuron's output has `mean=0` and `variance=1` AND is linearly decorrelated with the other neurons' outputs.

By 'linearly decorrelated', we mean that for a layer l with m units, individual unit activities $\mathbf{x} = \{\mathbf{x}^{(k)}, \ldots, \mathbf{x}^{(d)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \ldots \mathbf{x}^{(k)} \ldots \perp \mathbf{x}^{(d)}\}$. Note that we consider the unit activities to be random variables.

In short, we want to make sure that normalization/whitening for a single neuron's output is happening consistently across the entire dataset. In truth, this is not computationally feasible (you'd have to feed in the entire dataset at once), nor is it always fully differentiable. So instead, we maintain "running estimates" of the dataset's mean/variance and update them as we see more observations.

How do we do this? Remember that we're training on batches[1] - small groups of observations usually sized 16, 32, 64, etc. **Since each batch contains a random subsample of the dataset, we assume that each batch is somewhat representative of the entire dataset**. Based on this assumption, we can use their means and variances to update our running estimates.

B.2 Usage Notes

- Should I put BatchNorm before or after the activation function?
  - Before. Why? Google it! The community has settled on using it after the activation function, but there are strong arguments both ways.
- In general Dropout and BatchNorm don't work well together.
  - Why? See the abstract of this paper ; TLDR, Dropout messes up the variance that BatchNorm expects.
  - Depending on the problem space and even the researcher, they'll choose either Dropout or BatchNorm to use.
- BatchNorm works differently in training mode and evaluation mode. Depending on your current task, you must toggle these modes by running 'model.train()' or 'model.eval()', which will recursively set each layer to training or evaluation mode.

C Activation Functions

Here's a list of the various activations `torch` has available.

How do they differ? See this great guide[2]. While you can experiment, keep it safe at first - some activations, if misused, will completely break your network, and others (probably most) will make hardly noticeable contributions. You can get top performance on this assignment with only ReLU activations.

[1] Technically, *mini-batches*. "Batch" actually refers to the entire dataset. But colloquially and even in many papers, "batch" means "mini-batch".

[2] And keep it a secret that we linked another course's notes here. They're just great notes!

D Optimizers

D.1 Choosing an Optimizer

Here's a list of the various optimization algorithms you can use for stepping.

How do you choose and how do they differ? It gets complicated. Some of these will be discussed in lecture at some point.

But some general guidelines:

- For most single-GPU models, there are pretty much three optimizers that are commonly used: `SGD`, `Adam`, and `AdamW`.
- `SGD` is "old but gold"; it still works remarkably well. Convergence is a bit slower and more unstable than `Adam`, but this may actually be a good thing. A common opinion in DL research today is that `SGD` has better test performance than `Adam` because it finds more robust minima.
- `Adam` is an overwhelmingly popular choice of optimizer in modern DL research. It converges quickly to a good solution and is relatively stable.
- `AdamW` is a modification on `Adam` that attempts to 'fix' the way it handles **weight decay**, or L2 regularization. In practice, it seems to work very similarly to Adam, with maybe a slight improvement if you're using weight decay.

D.2 Choosing Hyperparams (e.g. LR, Momentum, Weight Decay, etc)

What are these hyperparameters, and how do you select them?

First, what are these? We'll cover these in lecture, but if anything's unclear, Google!

Second, how do you select them? The honest answer is 'What do other people use on this dataset' and trial-and-error. But some tips:

- If there's no precedent (i.e. you made this dataset or this is a private dataset for a company), you'll probably want to start from scratch
- If you start from scratch, you'll need to do some manual experimentation, usually with some kind of grid-search (i.e. try out various hyperparam combos, record which worked best). Alternatively, you could automatically find an LR using a package.
  - For automatic LR tuning, maybe Google something like 'pytorch lr finder'. There are several implementations, and they work on similar principles.

D.3 Schedulers

This is another can of worms. We won't talk about them in detail, but you can find a list of them here.

E Architecture Recommendations

Now for our recommendations. If you follow this, it should be enough to do well.

But if you want to improve your score further, read the above appendices, identify some trends, and try making changes in the directions they suggest.

E.1 Architecture

Assume `LBR(X, Y)` means `Linear(X, Y) -> BatchNorm1d(Y) -> ReLU`.

Try 5-10 `LBR` blocks, followed by an output `Linear` layer (no activation or BatchNorm afterwards). Here's an example of what you could try:

```
LBR(flattened_input_size, 1024) -> LBR(1024, 1024) -> LBR(1024, 1024) ->
   LBR (1024, 512) -> LBR (512, 256) -> LBR(256, 256) -> Linear(256, num_phonemes)
```

Remember, as memory allows, start by going as wide/deep as possible.

E.2 Everything Else

- `Adam`, with default learning rate of 10^{-3} should suffice. You should not need to go more than an order of magnitude above or below this (10^{-2}, 10^{-4}).
- Batch size of 1024, if it fits in memory. Anything above 256 and below 2048 should work. For now, assume the bigger the better.
- For most problems, the first two epochs are always the shakiest. But you can often start seeing good learning by the third epoch.
- You will not need more than 10 for most of your models. If you are using above 20, you may be wasting your time.
- If you do decide to use dropout instead of `BatchNorm1d`, use a probability between 0.1 and 0.4.

Optionally, you can explore learning rate schedulers, which helped in our experience! `StepLR` and `ReduceOnPlateau` are great options. One way of doing things is to decay by 0.95 each epoch, another is to decay 0.1 after little change has occurred. Feel free to experiment with other schedulers!