Creating a Text-To-Speech System in Rust

Daniel McKenna (xd009642)

Introduction

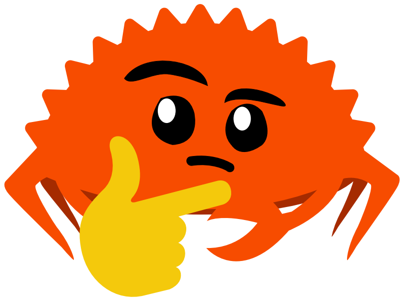
- Programmer at Emotech an AI startup primarily using Rust
- Primarily working in speech technologies and related areas
- xd009642 online
- May know me from cargo-tarpaulin

And This Talk?

- Introduce TTS systems and the challenges
- Cover all the stages of a pipeline
- Demonstrating it all with an open source TTS engine made for this talk!

Why Rust?

- Sometimes these AI systems need to be “real time”
- Also handle load from API users
- Python breaks down pretty quickly in this scenario
- Some researchers still create C++ based systems



What’s Hard About Text-To-Speed?

- Language is hard
  - Unknown words
  - Homographs: lead, bass, bow
  - Code-switching
- Speech is hard it has to sound natural - rhythm, tone, stress
- Naturalness conflicts with intelligibility
- Users want it controllable

How have we done TTS in the past?

- Formant Synthesis
- Concatenative Synthesis
- HMM Based Synthesis
- Deep learning
- And of course hybrid systems of the above

Formant Synthesis

- A formant is a resonance of the vocal tract
- Adding them together creates sounds
- By modelling how they change we can combine and make a sound
- Good intelligibility and runtime but sounds robotic
- Very low level modelling of speech so hard to develop

Concatenative Synthesis

- We have a database of audio samples for “units”
- Sub-word units e.g. syllables, phonemes, diphones
- We concatenate them to make audio
- Sounds natural except where the samples join there may be glitches

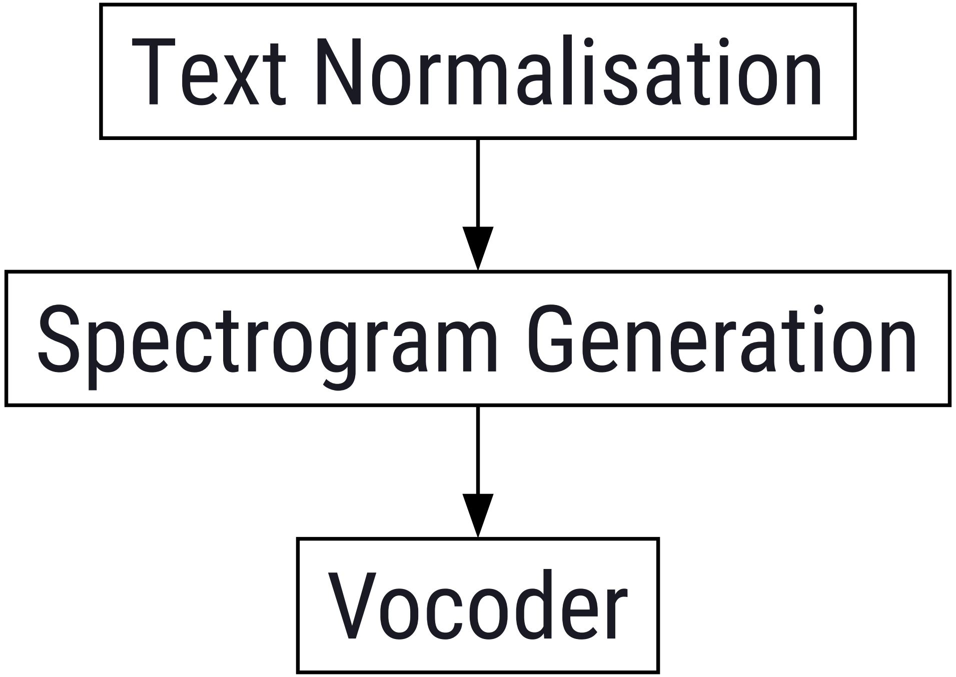
HMM Based Synthesis

- A statistical model of speech based on Hidden Markov Models
- Implementations typically use HTK - a C library
- Was state of the art pre-deep learning.
- Duration modelling is tricky!

Deep Learning

- Uses neural networks and a lot more data
- Typically one of 2 flavours:
  - Generates audio (end to end model)
  - Generates spectrogram then a vocoder (neural or otherwise) generates audio

Our System



What is Speech?

The Fundamentals

- We can break a word into a few different units:
  - Letters (graphemes)
  - Syllables - interpretted as a single sound
  - Phonemes - language specific sound that forms words
  - Phones - smallest unit of speech

Phonemes

- The words crab and cram are a single syllable
- They are multiple phonemes kɹæb vs kɹæm
- Using phonemes we can see what sounds are same and different
- IPA is a common phoneme alphabet but we'll use ARPABET
- CMU dict is an open source ARPABET dictionary

Advantage of Phonemes

- Traditionally we turn text into a smaller units
- Phonemes are a good element for this
- End to end deep learning systems sometimes won’t use them
- But this lacks control, we might mispronounce words

Is that it?

- No! As well as making the sounds correctly we want to model prosody
- Speech should have a natural intonation and rhythm
- This differs language to language.
- Languages can be stress, syllable (or mora) timed

Text Normalisation

Text Normalisation

- Convert text from written form to spoken form
- Was rule based but there are models for that
- A lot of people go for hybrid systems for customisation
- For our system we’re going to do a simpler rule based approach
- unicode segmentation and deunicode crates are great!

Challenges

- For the rules we need to identify to some level what each token is
- For example there’s a lot of ways to read out numbers like 1971
- Is a sequence of capital letters an initialism or shouting?
- Could we simplify normalisation and can we let users guide pronunciation?

SSML

- Speech Synthesis Markup Language an XML spec to guide a speech synthesiser
- Can use XML tags to give instructions to a TTS engine
- Best to build in support from day 1 - it can drive normalisation

Example SSML

```
<speak>
    I have <break time="3s"/>
    <say-as interpret-as='cardinal'>1</say-as>
    <phoneme alphabet='ipa' ph=''pi.kæn'>pecan</phoneme>
</speak>
```

Notable Rust Pattern!

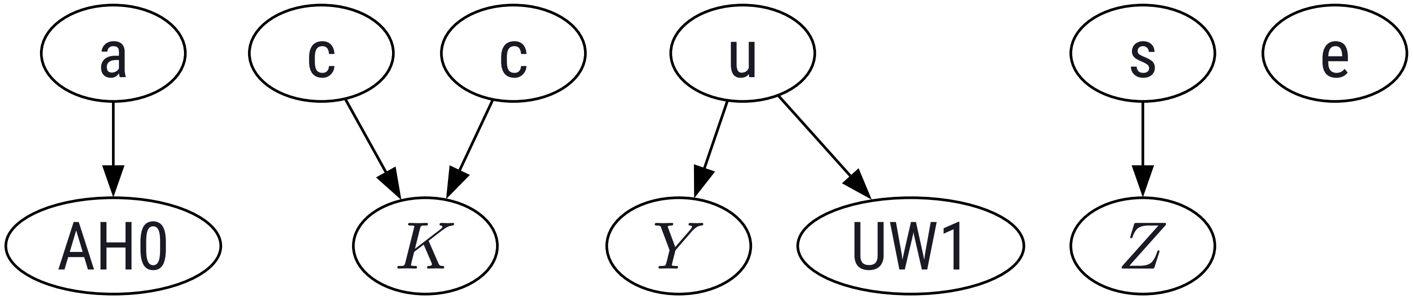
```rust
pub enum Element {
    Break,
}
pub enum ParsedElement {
    Break(BreakAttribute),
}
// The first one feels more normal to newcomers
parsed_element.tag() == Element::Break;
matches!(parsed_element, ParsedElement::Break(_));
// impl ParsedElement::tag is left as an exercise to reader
```

But We Still Have to Normalise

- Turn output into a list of chunks
- These are either: text, phonemes, tts state changes
- For text we split words, grab punctuation then normalise each word
- Keeping it simple (no context)!

The Final Step

- After normalisation often we turn words to phonemes
- For simplicity here we use a dictionary lookup approach
- For unseen words G2P (Grapheme to Phoneme) models are used.



Spectrogram Generation

Why Generate a Spectrogram?

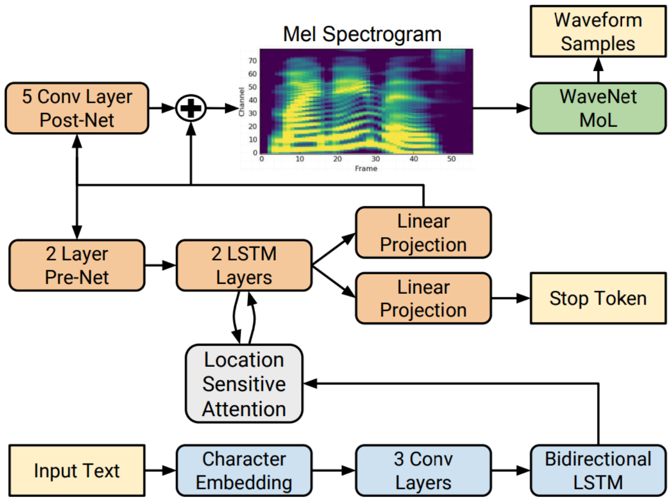
- The more we can constrain a problem the easier it is to train
- Higher dimensionality data requires more data to train
- So instead of generating audio we generate a simpler output

What Is a Mel Spectrogram?

- The mel scale is a pitch scale so that all tones sound equidistant
- For a window of time, this is like a histogram of frequency data
- The smaller a feature space the easier to fit a model
- Generating raw audio would require a lot more data

Tacotron2

- Sequence-to-sequence model, published 2018.
- No longer state of the art - but still very good



A Note on Neural Networks

- So here we’re going to avoid using Tensorflow or Torch
- Why? Because it’s more interesting (I hope)
- It also lets us look at more of the Rust Ecosystem including runtimes which can run on more devices

ONNX

- Open Neural Network Exchange
- A format to make it easier to run Neural Networks in any framework
- Adoption feels poor and ecosystem feels lacking
- But when it works it’s great
- Best native rust support is in tract
- ort is bindings to the official runtime (C++) and is fully featured

Useful ONNX Tools

- https://netron.app/ visualise the ONNX
- https://github.com/onnx/optimizer optimise the graphs for inference speed
- https://docs.nvidia.com/deeplearning/tensorrt/onnx-graphsurgeon/docs/index.html graph surgeon, introspect and manipulate ONNX graphs

ONNX and Tacotron2

- ONNX export splits the network into 3 subnetworks
- This is because of generally poor ONNX support in the ML ecosystem
- The ONNX export for default Tacotron2 vocoder doesn’t even succeed, it panics instead during export!

Tract

- Tract pure Rust and best spec support in the Rust ML ecosystem
- Missing loop blocks and dynamically sized inputs
- Optimising interpretter approach to ONNX
- Also inference speed isn’t competitive with non-Rust competitors
- Real Time Factor of 300 on “Hello world from Rust”

tract

Tract

```
type Model = SimplePlan<InferenceFact, Box<dyn InferenceOp>, Graph<InferenceFact,
Box<dyn InferenceOp>>>;

pub struct Tacotron2 {
   encoder: Model,
}

let encoder = tract_onnx::onnx()
    .model_for_path(path.as_ref().join("encoder.onnx"))?
    .into_runnable()?;

let phonemes = TValue::from_const(Arc::new(phonemes.into()));
let plen = Tensor::from_shape(&[1], &[phonemes.len() as i64])?;
let encoder_output = self.encoder.run(tvec![phonemes, plen])?;
```

ORT

- ONNX RunTime. Bindings to Microsoft's C++ ONNX runtime
- Best spec support in the wider ML ecosystem
- Decent performance - can perform optimisations
- Real Time Factor of 2.7 on “Hello world from Rust” (no optimisations)

ORT

```rust
pub struct Tacotron2 {
    encoder: Session,
}

let encoder = Session::builder()?
    .with_optimization_level(GraphOptimizationLevel::Level1)?
    .with_model_from_file(path.as_ref().join("encoder.onnx"))?;

let plen = arr1(&[phonemes.len() as i64]);
// also allows inputs!["phonemes"=> phonemes.view(), "plen" => plen.view()]
let encoder_outputs = self.encoder.run(inputs![phonemes, plen]?)?;
```

Thoughts

- ORT API has lower level components, but you can ignore them.
- But being able to specify inputs by name is really nice!
- Both have us using ndarray but tract forces wrapping it into their `Tensor` and `TValue` types
- Tract feels more idiomatic Rust and is easier to use, but `Tensor` vs `TValue` adds friction.

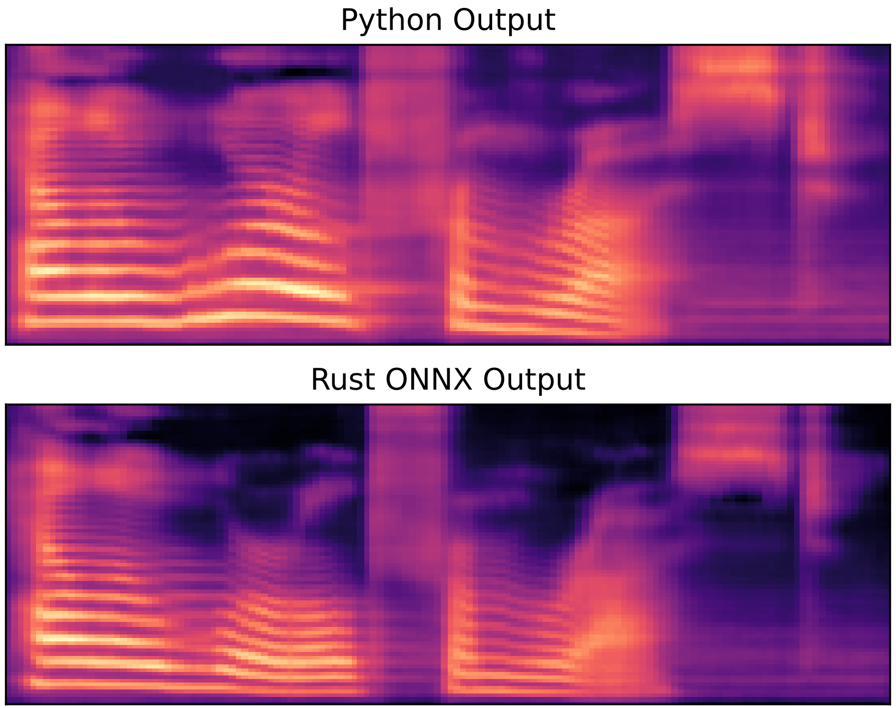
Why are Named Tensor Inputs/Outputs Useful?

```
let mut inputs = inputs![
    "decoder_input" => state.decoder_input.view(),
    "attention_hidden" => state.attention_hidden.view(),
    "attention_cell" => state.attention_cell.view(),
    "decoder_hidden" => state.decoder_hidden.view(),
    "decoder_cell" => state.decoder_cell.view(),
    "attention_weights" => state.attention_weights.view(),
    "attention_weights_cum" => state.attention_weights_cum.view(),
    "attention_context" => state.attention_context.view(),
    "memory" => memory.view(),
    "processed_memory" => processed_memory.view(),
    "mask" => state.mask.view()
]?;
```

Changes

- Three networks now
- We need to manually run the decoder iter keeping state
- The dynamic input dimension is now fixed because of JIT tracing
- The outputs between Python and Rust don’t look the same

But They Look Close!


Python Output
Rust ONNX Output

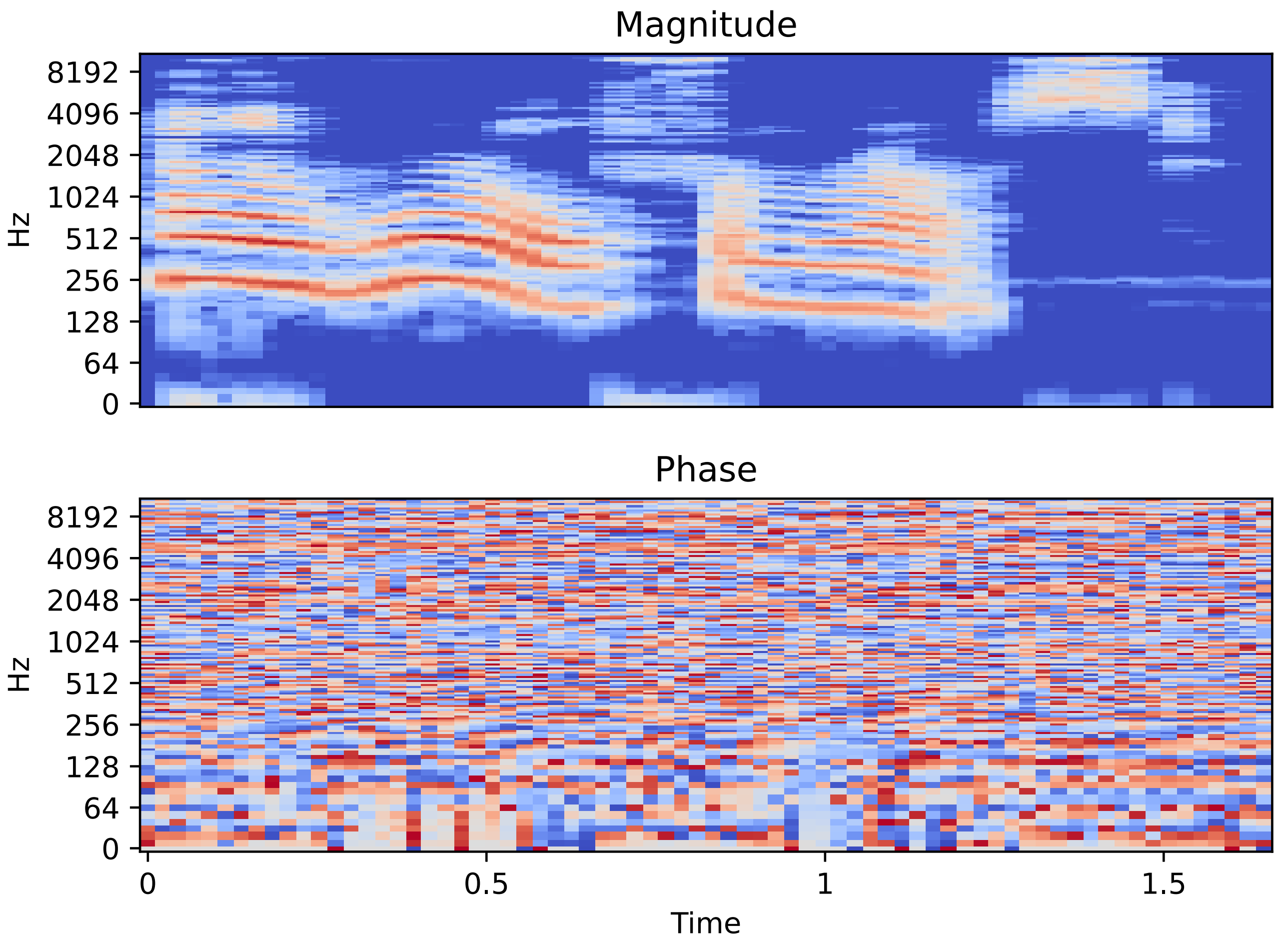
Don’t Trust Researcher Documentation

- Tacotron2's text processing says it can take uppercase/lowercase characters or ARPABET
- But the pretrained models weren’t trained with any ARPABET or uppercase characters
- You’ll get weird output!

The Vocoder

Turning It Into Sound

- In the frequency domain we have magnitude and phase information
- Magnitude is easier to learn
- Spectrogram generating techniques normally generate amplitude not phase
- We can get the parameters for vocoding from the Tacotron2 repo


Magnitude
8192
4096
2048
1024
512
256
128
64
0
Hz
Phase
8192
4096
2048
1024
512
256
128
64
0
Hz
0
0.5
1
1.5
Time

Griffin-Lim Basics

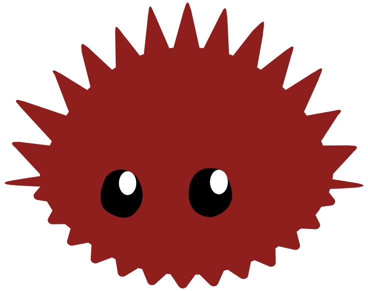
- Convert from mel spectrogram to linear spectrogram
- Create random phase spectrum
- Convert to audio and then back to spectrogram
- Restore the magnitude because *maths*.
- Repeat until stopping condition reached

How Do We Test It?

- We have a reference golden implementation
- Gather outputs from it and do a comparison
  - Comparing matrices of floats is a bit painful (lose developer UX)
- Testing with realistic inputs is the most valuable
- Aside from that learn and use unit testing to test your understanding

Notes on Implementation

- This was done as a port from librosa
- Wanted to compare with a well-understood analytic approach
- Never seen production, and while it’s tested it’s *less tested*

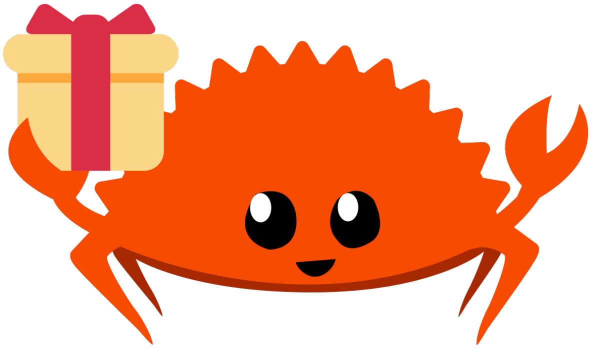


How Does it Sound?

- Griffin-Lim doesn’t have a model of how human speech sounds
- It just tries to do something simple and quick
- As a result there’s some artefacts

The Links!

- https://github.com/xd009642/xd-tts
- https://github.com/emotechlab/ssml-parser
- https://github.com/emotechlab/griffin-lim (plus tutorial)



Any Questions?