ELEC-C5220 Lecture 4: Virtual Analog modeling with Recurrent Neural Networks

Machine learning in information technology

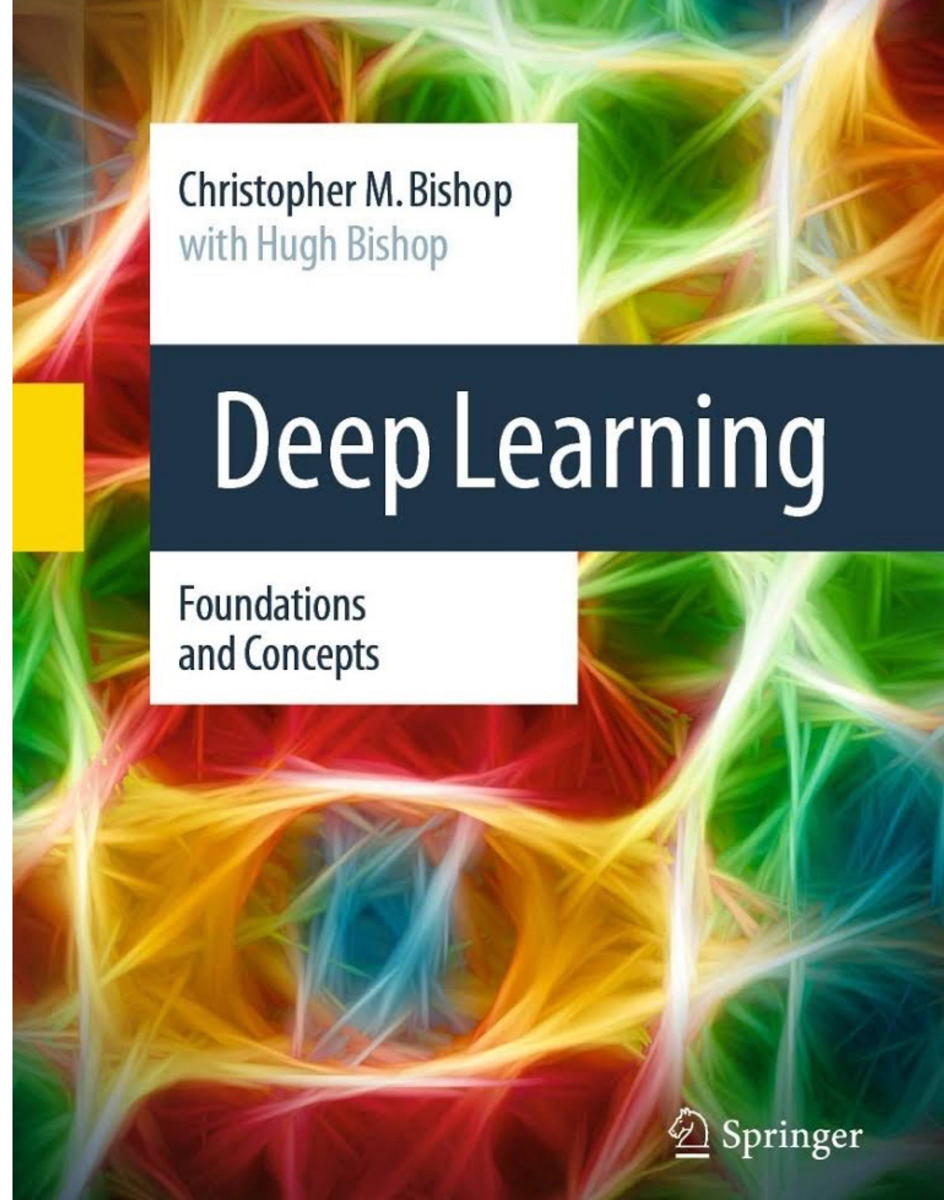
Aalto University
School of Electrical Engineering

Lauri Juvela

1.2.2024

A book recommendation

- Chris Bishop has a new book on Deep Learning! (2024)
- Not freely available, but this might be an official course book for next year
- Not required



Forward function in torch.nn.Module

```
input_dim = 10
output_dim = 3
batch_size = 5

# Create a random input tensor
x = torch.randn(batch_size, input_dim)

# Define a linear layer
layer = torch.nn.Linear(in_features=input_dim, out_features=output_dim)

# Three ways to call the forward method
y1 = layer(x)
y2 = layer.forward(x)
y3 = layer.__call__(x)
```

In-place functions in PyTorch

```
a = torch.ones(1)

# four ways to increment a by 1
a = a.add(1)
a.add_(1)
a = a + 1
a += 1

# one way to not increment a by 1
a.add(1)
```

Time limit in NGrader validation

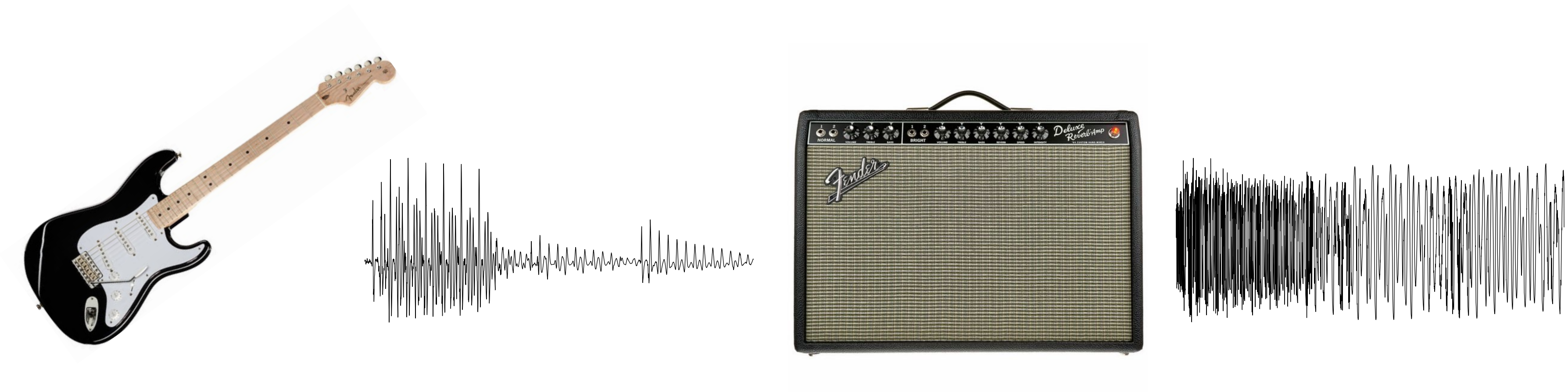
- **Some were getting timeouts when running Validate assignment**
    - Sometimes with useful error messages, sometimes not (invalid json character, etc.)
- **This is mostly a problem in Exercise 02, previous exercises are faster**
- **Limit was set to 4 minutes by default**
- **Increased validation time limit to 10 minutes since 31.1.2024**

Lecture 4 content

- Virtual analog modeling
- Recurrent neural networks
- Regression loss functions

Virtual Analog Modeling

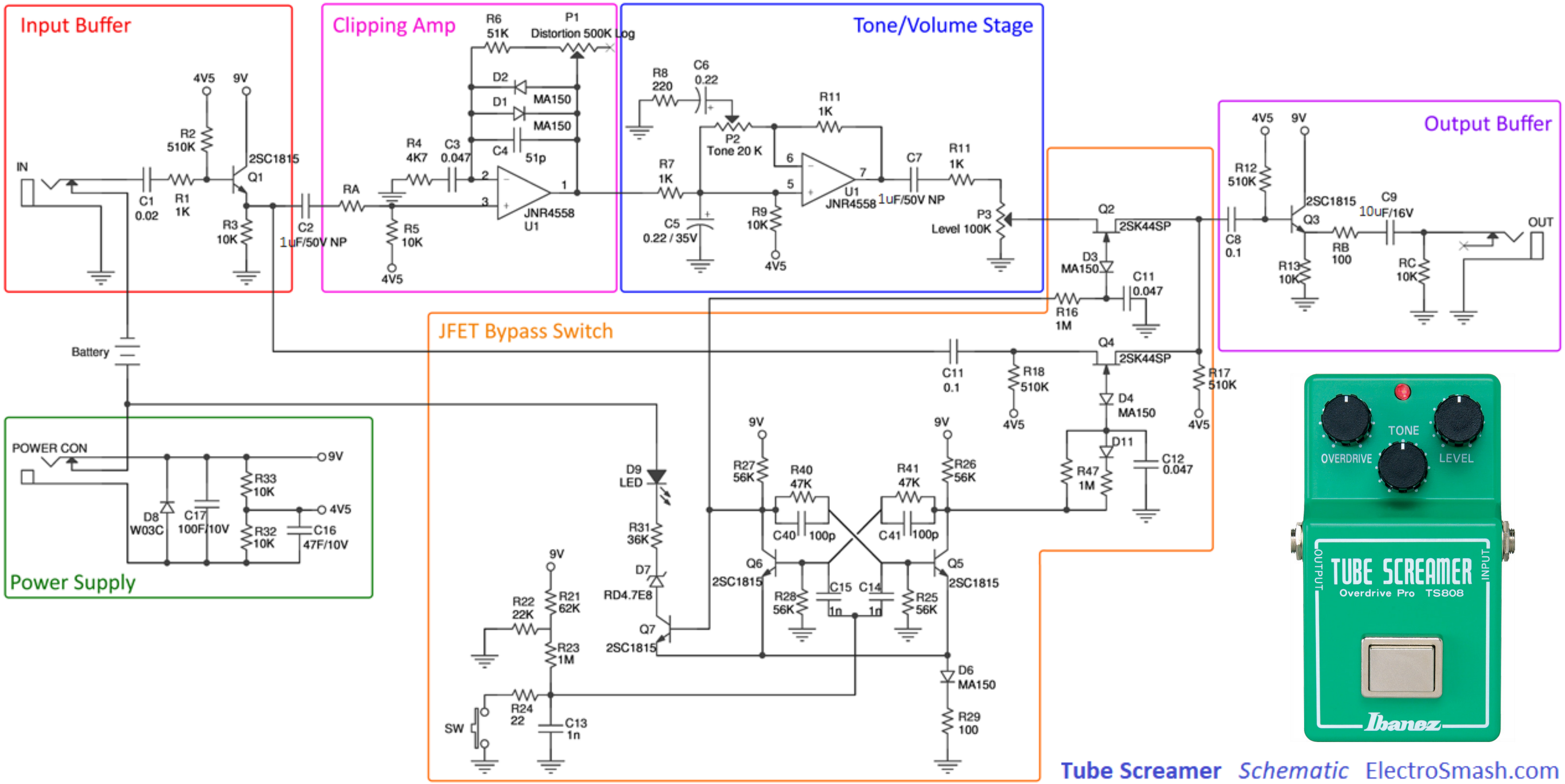
- **Replicate the tonal characteristics of analog audio effects in the digital domain**



White-box Virtual Analog Modeling

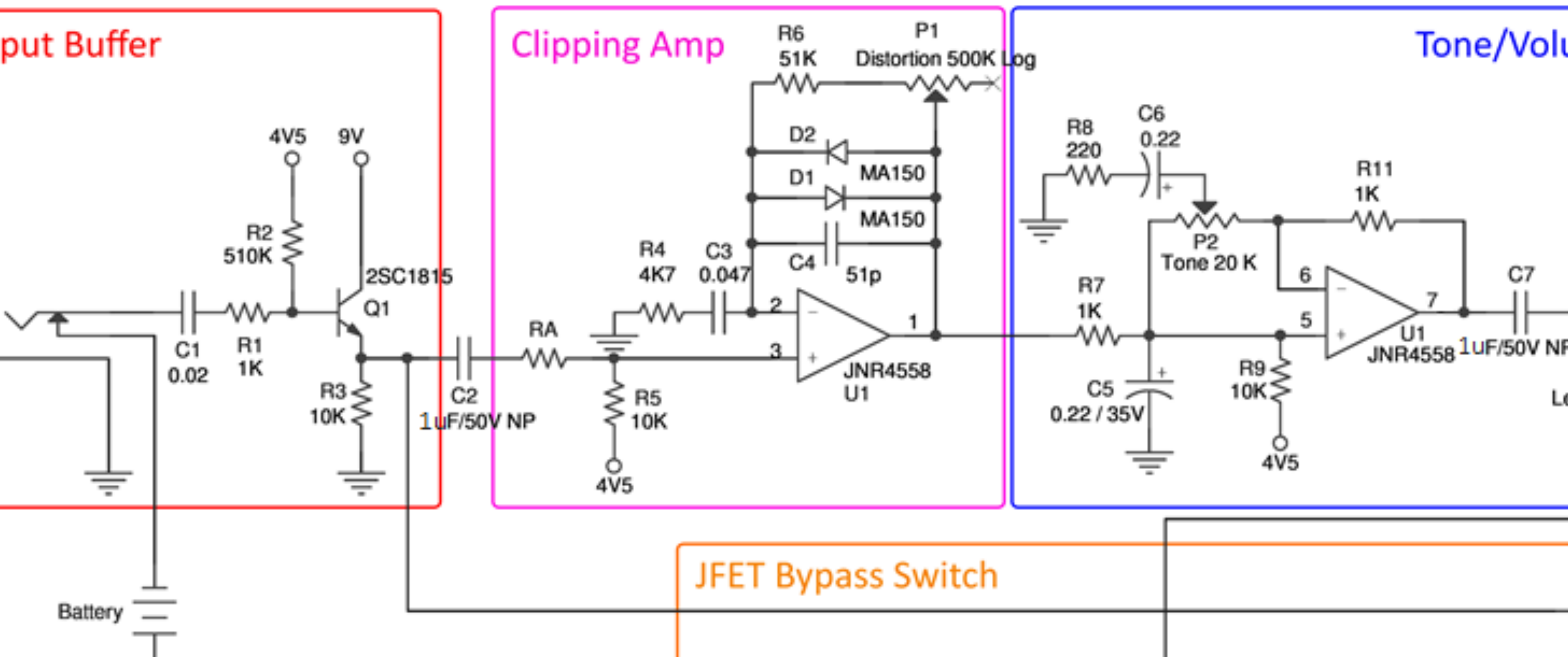
- Analyse analog circuit schematics
- Implement software emulation using digital signal processing (DSP) techniques
- If model runs too slowly for real-time, approximate
- Next: let’s look at some circuits to appreciate the problem
- No circuit analysis needed on this course

Tube Screamer schematic

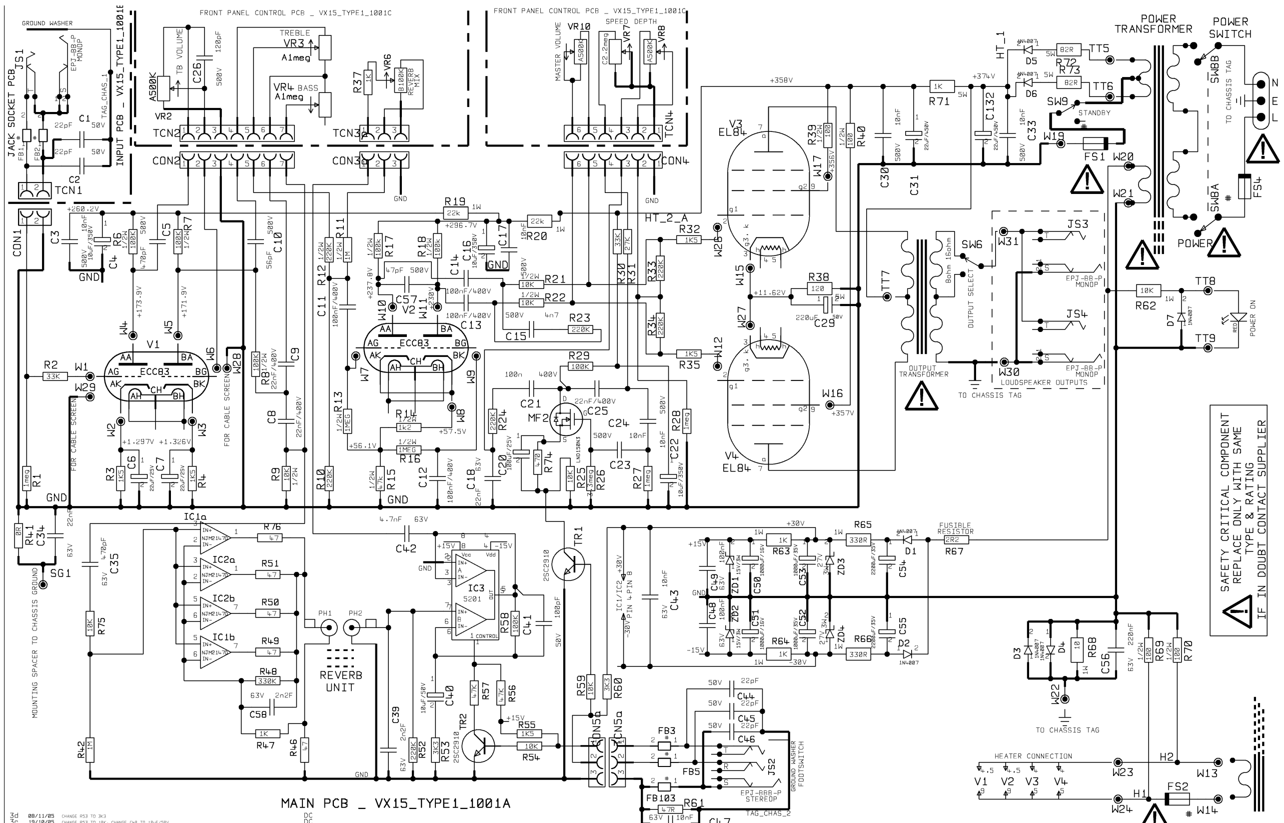


Aalto University
School of Electrical
Engineering

Tube Screamer clipping stage

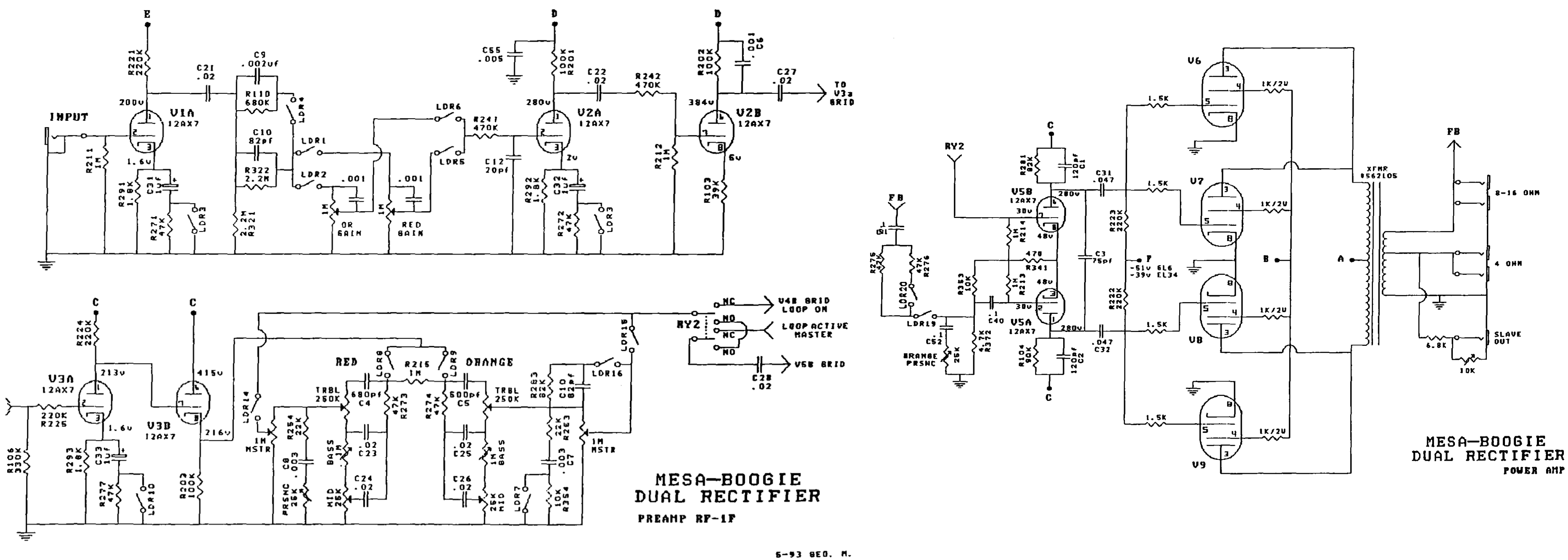


Vox AC15 Schematic



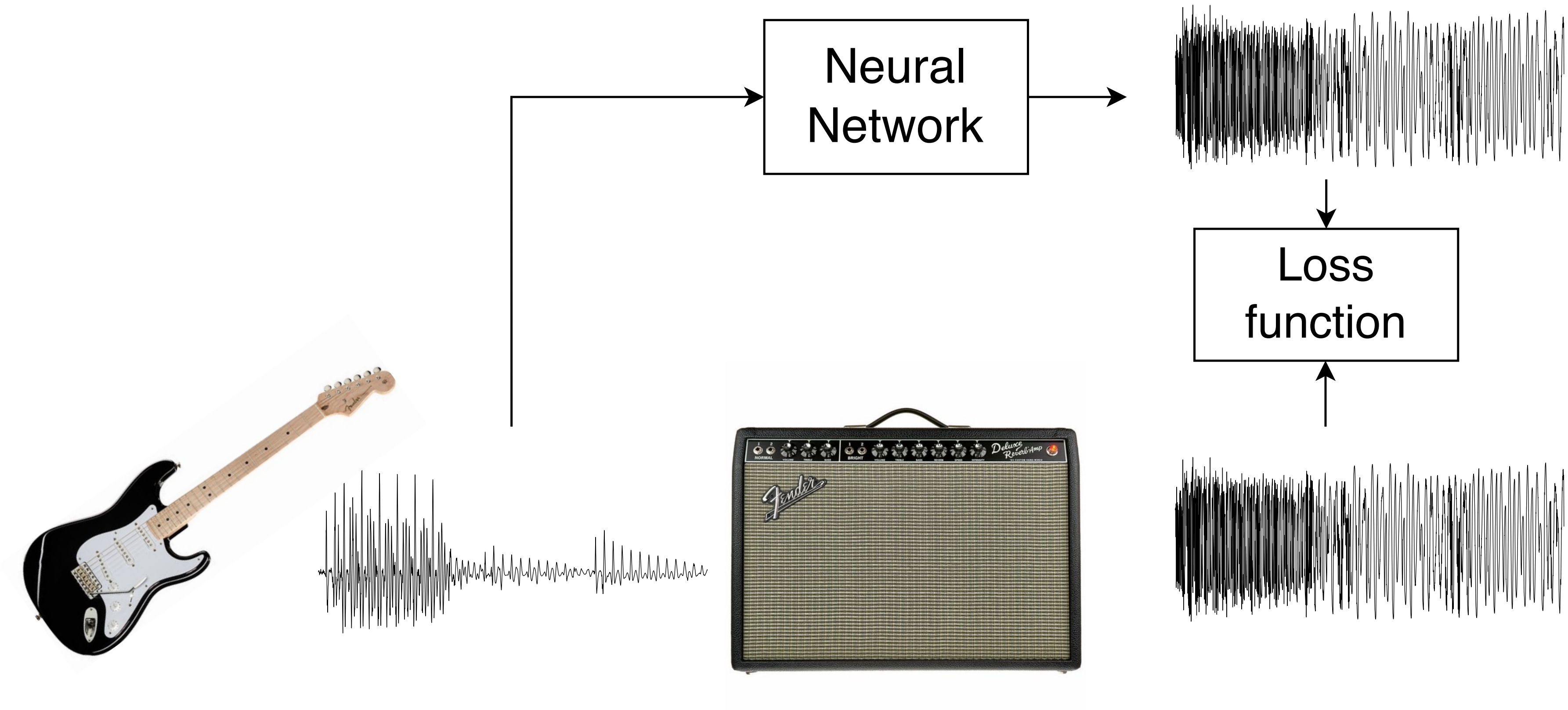
Aalto University
School of Electrical
Engineering

Mesa/Boogie Dual Rectifier schematic



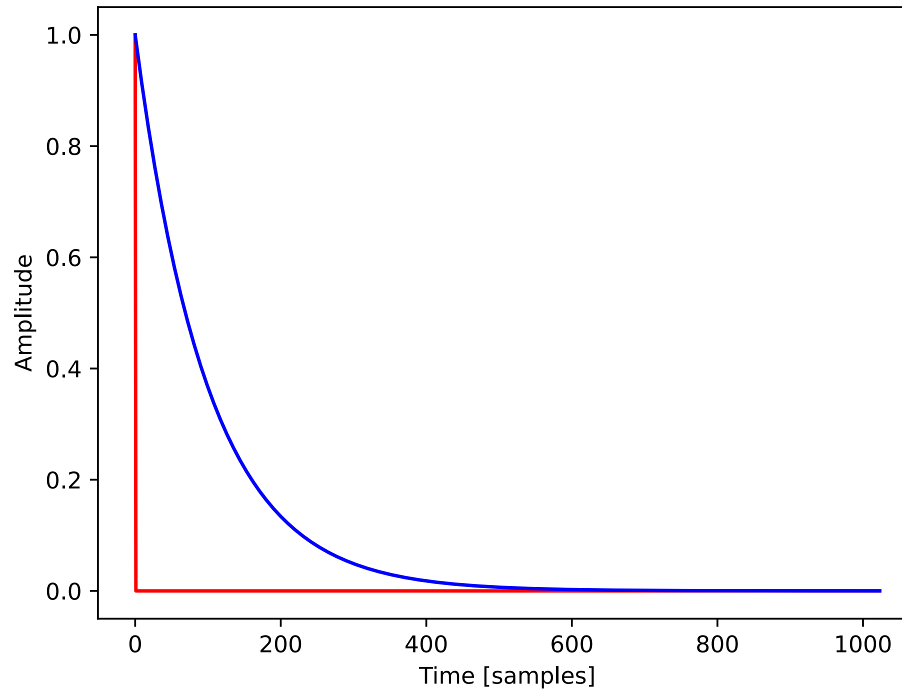
Aalto University
School of Electrical
Engineering

Black-box Virtual Analog Modeling



First order IIR filter

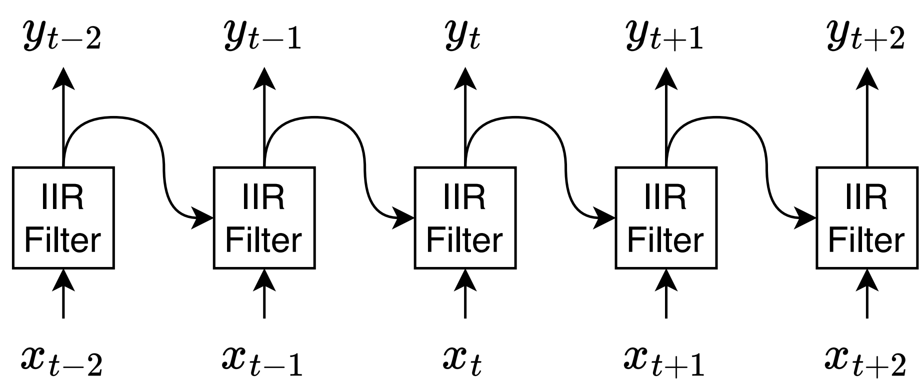
- **Adding feedback to filters makes them much more expressive**
- **First order Infinite Impulse Response (IIR)**
- **Exponential decay in impulse response**
- **Response can be approximated with FIR**



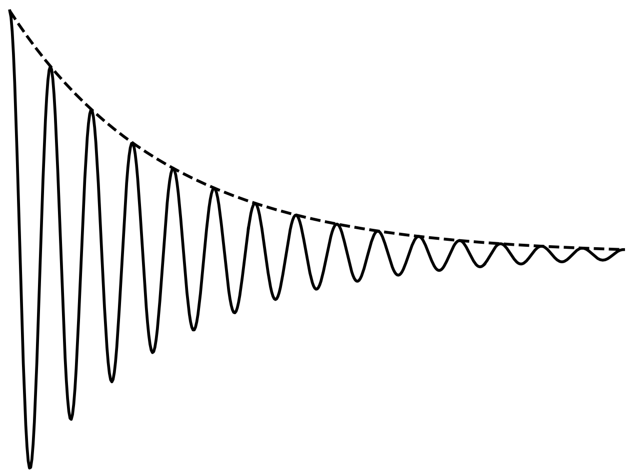
$$y_t = a_1 y_{t-1} + x_t$$

First order IIR unrolled in time

- For each time step, the filter output depends on the current input and previous state of the filter
- Apply the same operation on every time step (weight sharing)

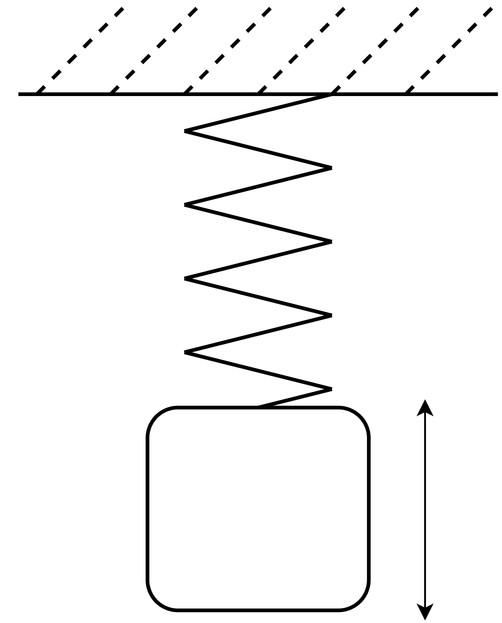


Damped oscillator



$$y(x) = e^{-\lambda x} \sin(\omega x + \phi)$$

damping oscillator

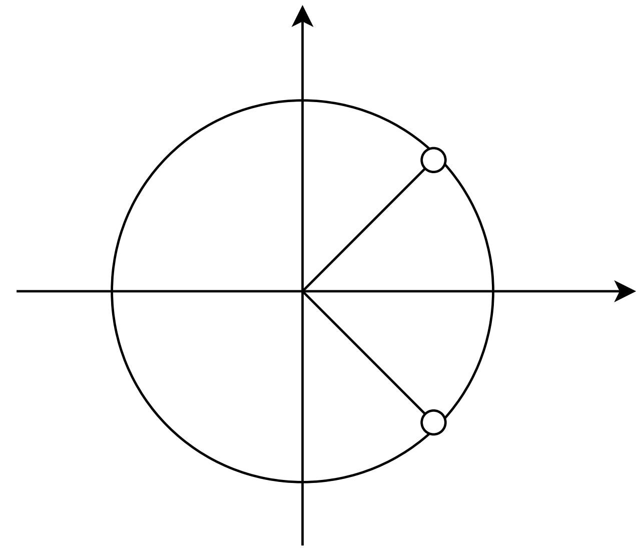


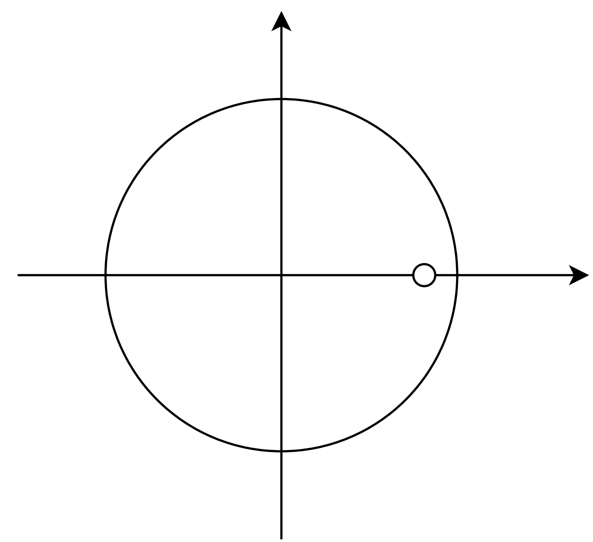
$$m \frac{\mathrm{d}^2}{\mathrm{d}t^2} x + c \frac{\mathrm{d}}{\mathrm{d}t} + kx = 0$$

mass damping spring

Damped oscillator with IIR filters

- Damping is exponential decay (first order IIR)
- Single frequency oscillation requires two filter poles placed on the unit circle (second orded IIR)





General IIR filter

- Filter output is a linear combination of current and previous input values, and previous output samples
- Expressive but still linear
- Parameter estimation is complicated

$$y_t = \sum_{i=1}^{P} a_i y_{t-i} + \sum_{j=0}^{M} b_j x_{t-j}$$

Recurrent Neural Networks

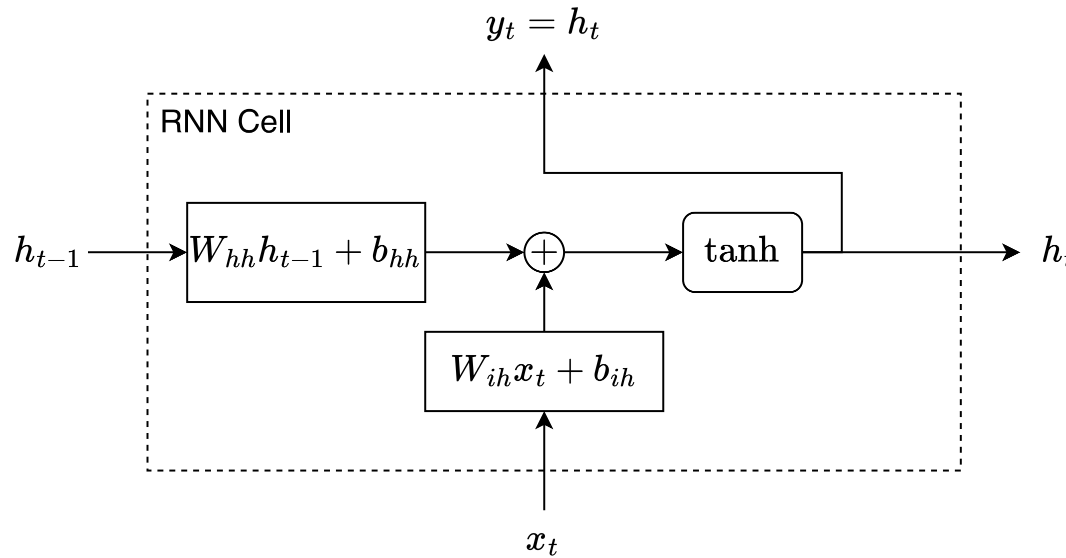
- Neural networks designed for time series processing
- A non-linear analogue of multi-channel first order IIR filters
- Related to state-space models and Markov chains
- RNN output at each time step depends on the current input, the previous state of the RNN, and the network

$$h_t = f(x_t, h_{t-1}; \theta)$$

Elman RNN, aka Vanilla RNN

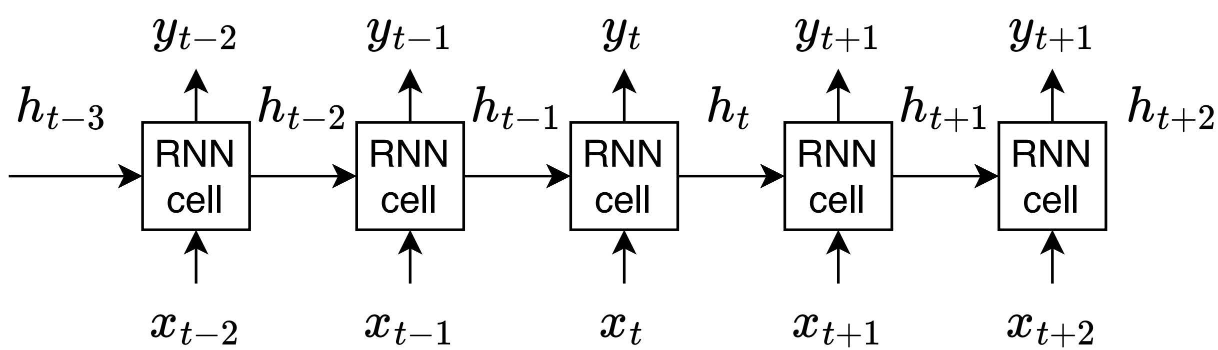
- **Output of the network is the same as the updated state**
- **Cell applies a Linear first order filtering step and saturating non-linearity**

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$



Unrolled RNNs

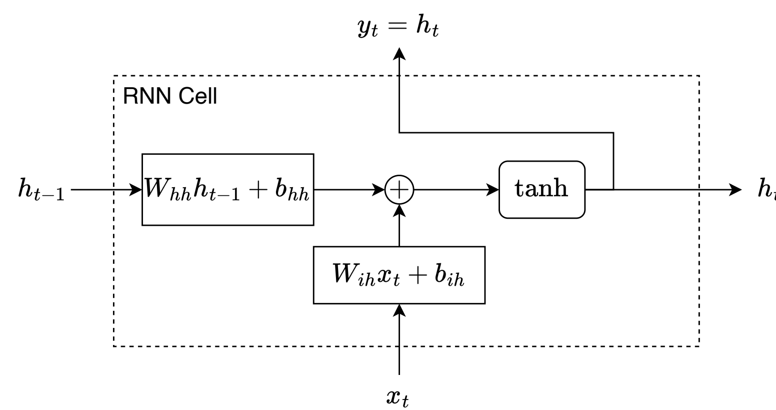
- **Forward pass requires sequential left-to-right processing**
- **Backward pass requires sequential right-to-left processing, aka backpropagation through time (BPTT)**
- **Network is deep in time and suffers from vanishing gradients**

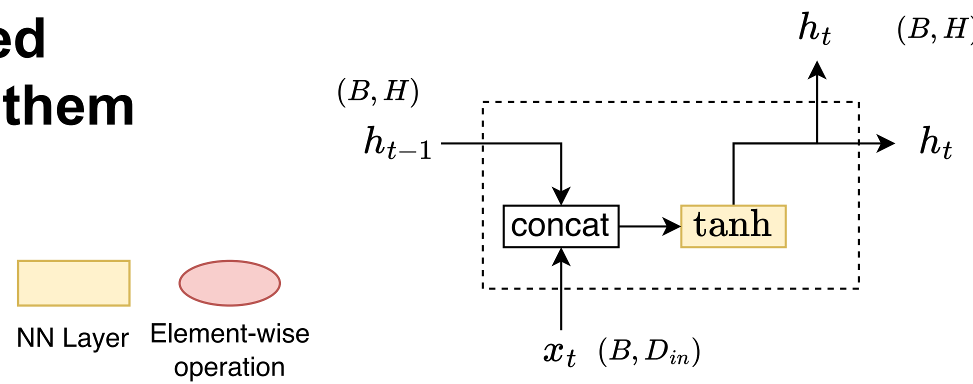


Graphical notation for RNNs

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$

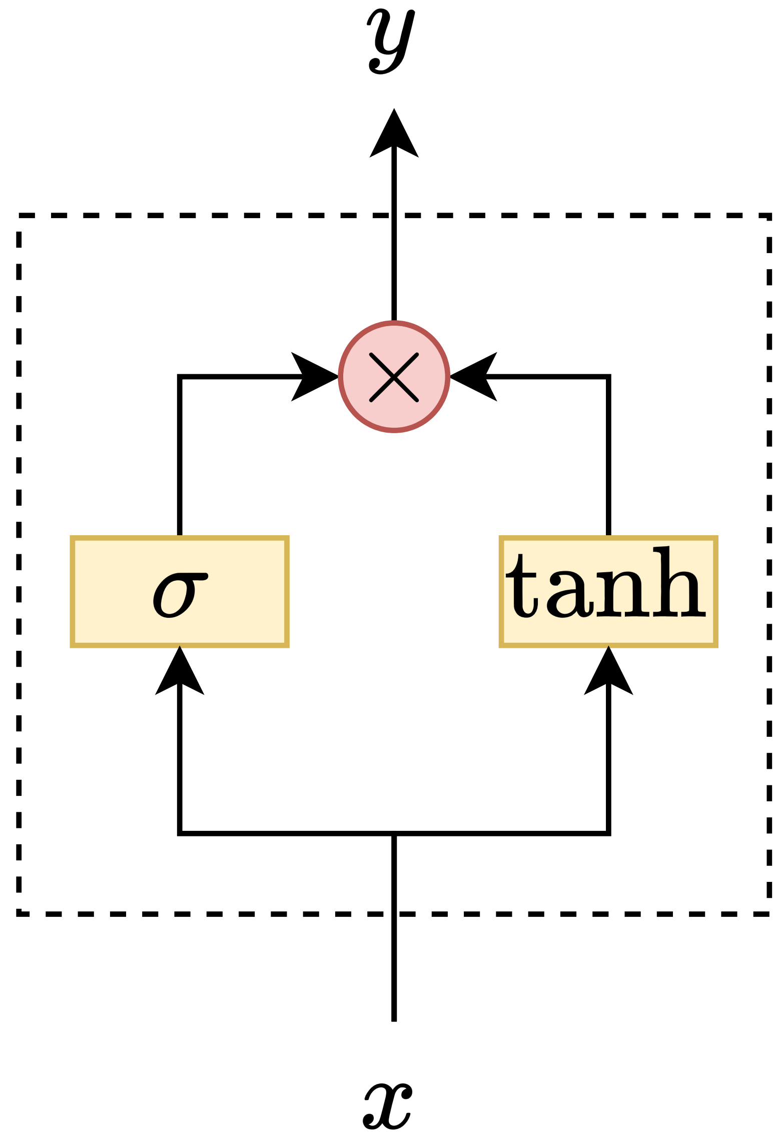
- Tanh layer includes weights that are correctly sized for the expected input
- In this case, the layer input size is H + D_{in}
- These are often called hidden cells, I'll call them hidden channels





Gated activation functions

- So far all the network structures we have seen have been additive
- Gated structures enable multiplication!
- Used extensively in RNNs, Gated CNNs (like WaveNet) and Attention in Tranformers



Gated RNNs

- Gated RNNs are designed for passing and modulating the network state through time to prevent issues with vanishing gradients
- Next: LSTM and GRU
- For more detailed analysis, see Chris Olah's blog post at https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)

- **LSTM Cell implements this set of equations**
- **Useful for programming not so intuitive for many**

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

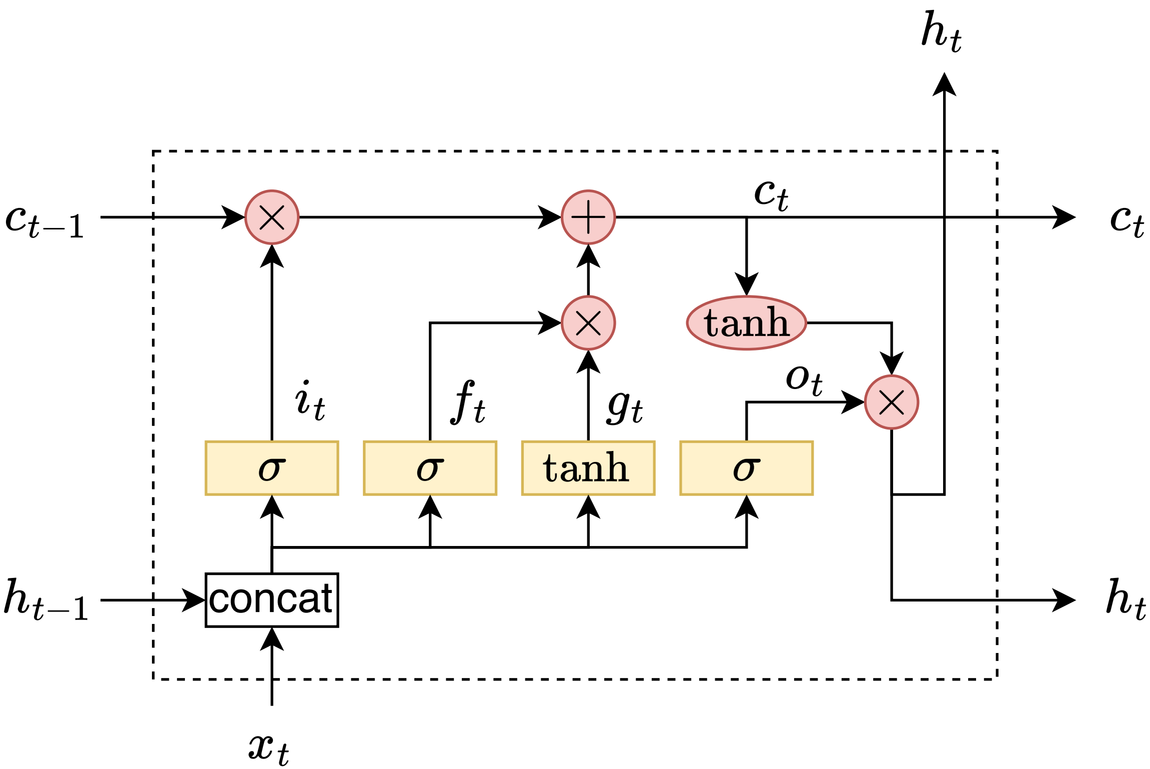
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)



$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

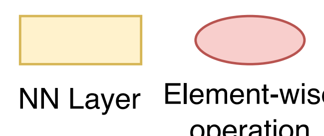
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$

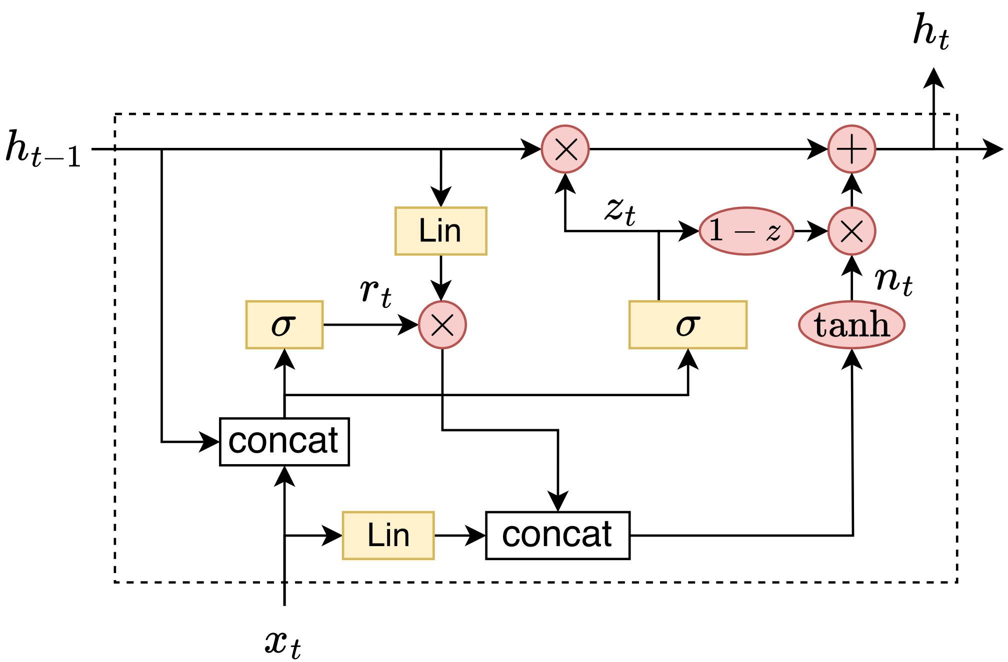
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$



Gated Recurrent Unit (GRU)



$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr})$$

$$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn}))$$

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz})$$

$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1}$$

RNN Layer vs RNN Cell

- RNN Layer processes as sequence in single forward call
- RNN Cell processes a single time step, you have to write a for loop over time
- Cells are useful for development and custom RNN design
- PyTorch has efficient implementations for LSTM and GRU layers that process the full sequence in a C++/CUDA backend without need to communicate with Python

LSTM in PyTorch

LSTM

CLASS torch.nn.LSTM(*self*, *input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *proj_size=0*, *device=None*, *dtype=None*) [SOURCE]

Apply a multi-layer long short-term memory (LSTM) RNN to an input sequence. For each element in the input sequence, each layer computes the following function:

LSTM in PyTorch

Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

LSTM in PyTorch

Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

LSTM in PyTorch

Outputs: output, (h_n, c_n)

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. When `bidirectional=True`, *output* will contain a concatenation of the forward and reverse hidden states at each time step in the sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the sequence. When `bidirectional=True`, *h_n* will contain a concatenation of the final forward and reverse hidden states, respectively.
- **c_n**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the sequence. When `bidirectional=True`, *c_n* will contain a concatenation of the final forward and reverse cell states, respectively.

A?

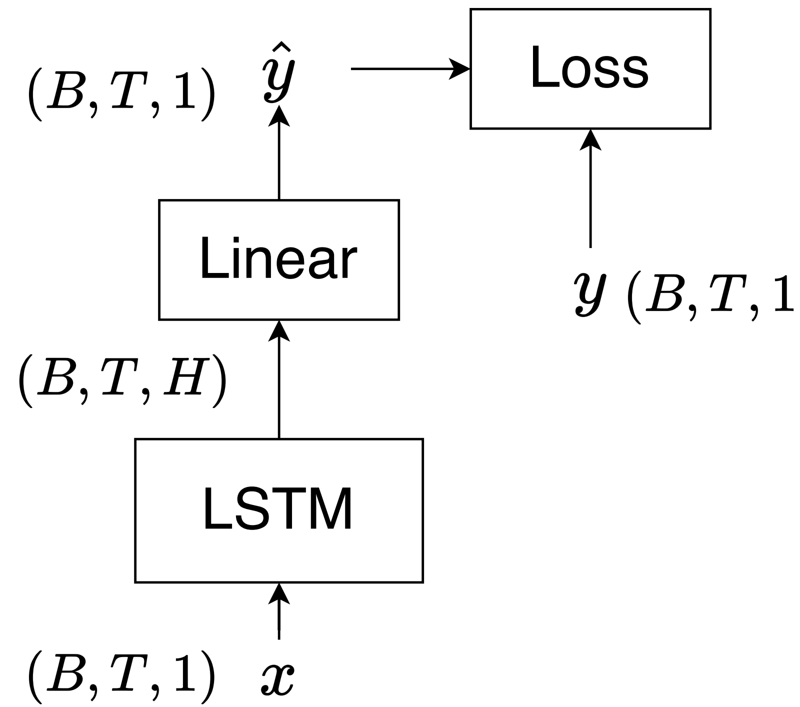
LSTM guitar amplifier model

- **One LSTM layer**
- **Linear layer projects from LSTM hidden dimension to**

B = Batch size

T = Timesteps

H = Number of hidden channels (cells)



Loss functions

- **Mean squared error (MSE) is the familiar L2 regression loss**
- **Error to Signal Ratio (ESR) normalises the error energy by signal energy**
- **Comparable to Signal-to-Noise Ratio (SNR)**

$$MSE(y,\hat{y}) = \frac{1}{BT}\sum_{B,T}(\hat{y}-y)^2$$

$$ESR(y,\hat{y}) = \frac{1}{BT}\frac{\sum_{B,T}(\hat{y}-y)^2}{\sum_{B,T}y^2}$$

Exercise this week

- Implement and test LSTM and GRU cells
- Implement and train a RNN guitar amp model

Lecture 4 summary

- Virtual analog modeling
- Recurrent neural networks
- PyTorch programming tips