Convolutional Codes

COS 463: Wireless Networks
Lecture 9
Kyle Jamieson

[Parts adapted from H. Balakrishnan]
Convolutional Coding: Motivation

• So far, we’ve seen block codes

• Convolutional Codes:
  – **Simple design**, especially at the transmitter
  
  – **Very powerful error correction** capability (used in NASA Pioneer mission deep space communications)
Convolutional Coding: Applications

- **Wi-Fi** (802.11 standard) and **cellular networks** (3G, 4G, LTE standards)

- Deep space **satellite communications**

- Digital Video Broadcasting (**Digital TV**)

- **Building block** in more advanced codes (**Turbo Codes**), which are in turn used in the above settings
1. Encoding data using convolutional codes
   – How the encoder works
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
Convolutional Encoding

- Don’t send message bits, send **only parity bits**
- Use a **sliding window** to select which message bits may participate in the parity calculations
**Sliding Parity Bit Calculation**

Message bits:

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 

\[ K = 4 \]

\[ P[0] = 0 \]

- Output: 0
Sliding Parity Bit Calculation

Message bits:

```
-3 -2 -1 0 1 2 3 4 5 6 7 8 .....  
0 0 0 0 1 1 0 1 0 0 1 0 1 1
```

\[ K = 4 \]

\[ P[1] = 1 \]

Output: 01
Sliding Parity Bit Calculation

$K = 4$

Message bits:

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 0 0 0 0 1 1 0 1 0 0 1 0 1

$P[2] = 0$

Output: 010
Sliding Parity Bit Calculation

Message bits:

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 0 0 0 0 1 1 0 1 0 0 1 0 1 1

K = 4

P[3] = 1

Output: 0100
Multiple Parity Bits

Message bits:

\[ \text{Output: \ldots 11} \]

\[ P_2[3] = 1 \]

\[ P_1[3] = 1 \]
Multiple Parity Bits

Message bits:

0 0 0 0 1 1 0 1 0 0 1 0 1

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 

\[ P_2[4] = 0 \]

\[ P_1[4] = 0 \]

Output: ....1100
Multiple Parity Bits

Message bits:

-3 -2 -1 0 1 2 3 4 5 6 7 8 ..... 0 0 0 0 1 1 0 1 0 0 1 0 1

P₂[5] = 1

P₁[5] = 0

Output: ….110001
Encoder State

- **Input bit** and **K-1 bits of current state** determine state on next clock cycle
  - Number of states: $2^{K-1}$
Constraint Length

• $K$ is the constraint length of the code

• Larger $K$:
  – Greater redundancy
  – Better error correction possibilities (usually, not always)
Transmitting Parity Bits

• Transmit the parity sequences, not the message itself
  – Each message bit is “spread across” K bits of the output parity bit sequence

  – If using multiple generators, **interleave** the bits of each generator
    • e.g. (two generators):

\[
p_0[0], p_1[0], p_0[1], p_1[1], p_0[2], p_1[2]
\]
Transmitting Parity Bits

- **Code rate** is $1 / \#_{\text{of generators}}$
  - e.g., 2 generators $\rightarrow$ rate = $\frac{1}{2}$

- **Engineering tradeoff:**
  - More generators *improves bit-error correction*
    - But *decreases rate of the code* (the number of message bits/s that can be transmitted)
Shift Register View

- One message bit $x[n]$ in, two parity bits out
  - **Each timestep:** message bits shifted right by one, the incoming bit moves into the left-most register

The values in the registers define the state of the encoder
**0th stream:** \( p_0[n] = x[n] + x[n-1] + x[n-2] \) (mod 2)

**1st stream:** \( p_1[n] = x[n] + x[n-2] \) (mod 2)

*The values in the registers define the state of the encoder*
Today

1. Encoding data using convolutional codes
   – Encoder state machine
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
State Machine View

- Example: \( K = 3 \), code rate = \( \frac{1}{2} \), convolutional code
  - There are \( 2^{K-1} \) states
  - **States** labeled with \((x[n-1], x[n-2])\)
  - **Arcs** labeled with \(x[n]/p_0[n]p_1[n]\)
  - **Generator:** \( g_0 = 111, g_1 = 101 \)
  - \( \text{msg} = 101100 \)
State Machine View

• \( P_0[n] = (1 \times x[n] + 1 \times x[n-1] + 1 \times x[n-2]) \mod 2 \)
• \( P_1[n] = (1 \times x[n] + 0 \times x[n-1] + 1 \times x[n-2]) \mod 2 \)
• Generators: \( g_0 = 111, g_1 = 101 \)

• \( \text{msg} = 101100 \)
• Transmit:
State Machine View

- P₀[n] = 1*1 + 1*0 + 1*0 mod 2
- P₁[n] = 1*1 + 0*0 + 1*0 mod 2
- Generators: g₀ = 111, g₁ = 101

- msg = 101100
- Transmit: 11
State Machine View

- $P_0[n] = 1*0 + 1*1 + 1*0 \mod 2$
- $P_1[n] = 1*0 + 0*1 + 1*0 \mod 2$
- Generators: $g_0 = 111, g_1 = 101$

- $msg = 101100$
- Transmit: 11 10
State Machine View

Starting state

- $P_0[n] = 1*1 + 1*0 + 1*1 \mod 2$
- $P_1[n] = 1*1 + 0*0 + 1*1 \mod 2$
- Generators: $g_0 = 111, g_1 = 101$

- msg = 101100
- Transmit: 11 10 00
State Machine View

- **Starting state**
  - \( P_0[n] = 1\times 1 + 1\times 1 + 1\times 0 \)
  - \( P_1[n] = 1\times 1 + 0\times 1 + 1\times 0 \)
  - **Generators**: \( g_0 = 111, g_1 = 101 \)

- **msg** = 101100
- **Transmit**: 11 10 00 01
State Machine View

Starting state

- $P_0[n] = 1*0 + 1*1 + 1*1$
- $P_1[n] = 1*0 + 0*1 + 1*1$
- Generators: $g_0 = 111$, $g_1 = 101$

- $msg = 101100$
- Transmit: 11 10 00 01 01
State Machine View

- \( P_0[n] = 1 \times 0 + 1 \times 0 + 1 \times 1 \)
- \( P_1[n] = 1 \times 0 + 0 \times 0 + 1 \times 1 \)
- Generators: \( g_0 = 111 \), \( g_1 = 101 \)

- msg = 101100
- Transmit: 11 10 00 01 01 11
Today

1. Encoding data using convolutional codes
   - How the encoder works
   - Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
Varying the Code Rate

- **How to increase the rate** of a convolutional code?

- Transmitter and receiver agree on coded bits to omit
  - *Puncturing table* indicates which bits to include (1)
    - Contains $p$ rows (one per parity equation), $N$ columns

- Example:
Punctured convolutional codes: example

- Coded bits =

```
0 0 1 0 1
0 0 1 1 1
```

- With Puncturing:

```
\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]
```

Puncturing table
Punctured convolutional codes: example

- Coded bits =

```
0 0 1 0 1
0 0 1 1 1
```

- With Puncturing matrix:

\[ P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \]

5 out of 8 bits are retained
Punctured convolutional codes: example

- Coded bits =

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 \\
\end{array}
\]

- With Puncturing matrix:

\[
P_1 = \begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
\end{pmatrix}
\]

- Punctured, coded bits:

\[
\begin{array}{c}
0 \\
0 \\
\end{array}
\]
Punctured convolutional codes: example

- Coded bits =

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
\end{array}
\]

- With Puncturing matrix:

\[
P_1 = \begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
\end{pmatrix}
\]

- Punctured, coded bits:

\[
\begin{array}{cc}
0 & 0 \\
0 & 0 \\
\end{array}
\]
Punctured convolutional codes: example

- Coded bits =

```
0 0 1 0 1
0 0 1 1 1
```

- With Puncturing matrix:

\[
P_1 = \begin{pmatrix}
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1
\end{pmatrix}
\]

- Punctured, coded bits:

```
0 0 1
0 0 1
```
Punctured convolutional codes: example

• Coded bits =

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
\end{array}
\]

• With Puncturing matrix:

\[
P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}
\]

• Punctured, coded bits:

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
\end{array}
\]
Punctured convolutional codes: example

• Coded bits =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• With Puncturing matrix:

\[
P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}
\]

• Punctured, coded bits:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Punctured convolutional codes: example

• Coded bits =

```
0 0 1 0 1
0 0 1 1 1
```

• Punctured, coded bits:

```
0 0 1 1
0 1 1
```

• Punctured rate is increased to: \( R = \frac{1/2}{5/8} = \frac{4}{5} \)
• Consider a convolutional code whose parity equations are:

\[ p_0 = x[n] + x[n - 1] + x[n - 3] \]
\[ p_1 = x[n] + x[n - 1] + x[n - 2] \]
\[ p_2 = x[n] + x[n - 2] + x[n - 3] \]

1. What’s the rate of this code? How many states are in the state machine representation of this code?

2. To increase the rate of the given code, 463 student Lem E. Tweakit punctures it with the following puncture matrix:

\[
\begin{pmatrix}
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

What’s the rate of the resulting code?
Today

1. Encoding data using convolutional codes

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard decision decoding
   – Soft decision decoding
Motivation: The Decoding Problem

- Received bits: **000101100110**
- Some errors have occurred
- *What’s the 4-bit message?*
- **Most likely: 0111**
  - Message whose coded bits is *closest to received bits* in Hamming distance

<table>
<thead>
<tr>
<th>Message</th>
<th>Coded bits</th>
<th>Hamming distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000000000000</td>
<td>5</td>
</tr>
<tr>
<td>0001</td>
<td>000000111011</td>
<td>6</td>
</tr>
<tr>
<td>0010</td>
<td>000011101100</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>000011010111</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>001110110000</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>001110001011</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>001101011100</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>001101100111</td>
<td>2</td>
</tr>
<tr>
<td>1000</td>
<td>110110000000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>110111111011</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>11000101100</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>110000101111</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>110101110000</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>110100101111</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>110110011100</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>110110100111</td>
<td></td>
</tr>
</tbody>
</table>
The Trellis

- Vertically, lists encoder states
- Horizontally, tracks time steps
- Branches connect states in successive time steps

Trellis:

States

0 0
0 1
1 0
1 1

Time →
x[n-1] x[n-2]
The Trellis: Sender’s View

- At the sender, transmitted bits trace a unique, single path of branches through the trellis
  - e.g. transmitted data bits 1 0 1 1

- Recover transmitted bits ⇔ Recover path
Viterbi algorithm

- **Want**: Most likely sent bit sequence

- Calculates *most likely path* through *trellis*

1. **Hard input Viterbi algorithm**: Have possibly-corrupted encoded *bits*, after reception

2. **Soft input Viterbi algorithm**: Have possibly-corrupted *likelihoods* of each bit, after reception
   - e.g.: “this bit is 90% likely to be a 1.”
Viterbi algorithm: Summary

- **Branch metrics** score *likelihood of each trellis branch*

- At any given time there are $2^{K-1}$ *most likely messages* we’re tracking (one for each state)
  - One message ↔ one trellis path
  - **Path metrics** score *likelihood of each trellis path*

- **Most likely message** is the one that produces the *smallest* path metric
1. Encoding data using convolutional codes

2. Decoding convolutional codes: Viterbi Algorithm
   - Hard input decoding
   - Soft input decoding
Hard-input branch metric

- Hard input $\rightarrow$ input is bits

- **Label every branch** of trellis with branch metrics
  - *Hard input Branch metric*: **Hamming Distance** between received and transmitted bits

Received: 00

<table>
<thead>
<tr>
<th>States</th>
<th>0 0</th>
<th>0 1</th>
<th>1 0</th>
<th>1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch metrics:
- $0/00 \rightarrow 0$
- $1/11 \rightarrow 2$
- $0/11 \rightarrow 2$
- $0/01 \rightarrow 1$
- $1/00 \rightarrow 0$
- $1/01 \rightarrow 1$
- $0/10 \rightarrow 1$
- $1/10 \rightarrow 1$
Hard-input branch metric

- Suppose we know encoder is in state 00, receive bits: 00
Hard-input path metric

- **Hard-input path metric:** Sum Hamming distance between sent and received bits along path

- Encoder is initially in state 00, receive bits: 00

```
Received:       00
```

![Diagram](image)
Hard-input path metric

- Right now, each state has a **unique** predecessor state
- Path metric: Total bit errors **along path ending at state**
  - Path metric of predecessor + branch metric

Received: 00 11

```
0 0 ——— 0/00 → 0 ——— 0/00 → 2 ——— 2
/00 ——— 1/11 → 2 ——— 0/10 → 1 ——— 3
0 1 ——— 1/11 → 0 ——— 1/01 → 1 ——— 3
1 0 ——— 1/01 → 1 ——— 0 ——— 0
1 1 ——— 1/01 → 1 ——— 0 ——— 0
```
Each state has two predecessor states, two *predecessor paths* (which to use?)

- **Winning** branch has *lower* path metric (*fewer* bit errors): *Prune* losing branch
Hard-input path metric

- Prune losing branch for each state in trellis

Received: 00 11 01
**Pruning non-surviving branches**

- **Survivor path** begins at each state, traces **unique path** back to **beginning** of trellis
  - **Correct path** is one of **four** survivor paths

- Some branches are not part of any survivor: **prune them**

![Diagram showing trellis and survivor paths with arrows indicating transitions and pruning of non-surviving branches.](image)
• When only one branch remains at a stage, the Viterbi algorithm decides that branch’s input bits:

```
Received:  00  11  01
Decide:    0
```

![Diagram showing the Viterbi algorithm process with received bits and decision points.](Diagram)

- Received bits: 00, 11, 01
- Decide: 0
- Diagram shows the transition from received bits to decision points through state transitions.
End of received data

- **Trace back** the survivor with **minimal path metric**
- Later stages **don’t get benefit** of future error correction, had data not ended

<table>
<thead>
<tr>
<th>Received:</th>
<th>00</th>
<th>11</th>
<th>01</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decide:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0 0
0 1
1 0
1 1
Terminating the code

- **Sender** transmits *two 0 data bits* at end of data

- **Receiver** uses the following trellis at end:

  ![Trellis Diagram]

  - After termination only one trellis survivor path remains
    - Can *make better bit decisions at end of data* based on this sole survivor
Viterbi with a Punctured Code

- Punctured bits are never transmitted

- Branch metric measures dissimilarity only between received and transmitted unpunctured bits
  - Same path metric, same Viterbi algorithm
  - Lose some error correction capability
Today

1. Encoding data using convolutional codes

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard input decoding
     • Error correcting capability
   – Soft input decoding
How many bit errors can we correct?

- Think back to the encoder; **linearity property:**
  - Message $m_1 \to$ Coded bits $c_1$
  - Message $m_2 \to$ Coded bits $c_2$
  - Message $m_1 \oplus m_2 \to$ Coded bits $c_1 \oplus c_2$

- So, $d_{\text{min}} = \text{minimum distance between 000...000 codeword and codeword with fewest 1s}$
Calculating $d_{\text{min}}$ for the convolutional code

- Find path with **smallest non-zero path metric** going from first 00 state to a future 00 state

- Here, $d_{\text{min}} = 4$, so can correct 1 error in 8 bits:

- The free distance is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. It is 4 in this example. The path 00/10/01/00 has a shorter length, but a higher path metric (of 5), so it is not the free distance.

Figure 8-6: The free distance of a convolutional code.
Today

1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. Decoding convolutional codes: Viterbi Algorithm
   – Hard input decoding
   – Soft input decoding
Model for Today

- **Coded bits** are actually **continuously-valued “voltages”** between 0.0 V and 1.0 V:

  - **Strong “0”**
  - **Strong “1”**
  - **Weak “0”**
  - **Weak “1”**
On Hard Decisions

• Hard decisions digitize each voltage to “0” or “1” by comparison against *threshold voltage 0.5 V*
  – Lose information about how “good” the bit is
    • Strong “1” (0.99 V) treated equally to weak “1” (0.51 V)

• **Hamming distance** for branch metric computation

• But *throwing away information* is almost never a good idea when making decisions
  – Find a *better branch metric that retains information about the received voltages?*
Soft-input decoding

- **Idea:** Pass received voltages to decoder before digitizing
  - **Problem:** Hard branch metric was Hamming distance

- **“Soft” branch metric**
  - **Euclidian distance** between received voltages and voltages of expected bits:

  - Expected parity bits: (0, 1)
Soft-input decoding

- **Different** branch metric, hence **different** path metric
  - **Same** path metric computation

- **Same** Viterbi algorithm

- **Result:** Choose **path** that minimizes sum of squares of Euclidean **distances** between received, expected voltages
Putting it together: Convolutional coding in Wi-Fi

Data bits

Convolutional encoder

Coded bits

Modulation (BPSK, QPSK, …)

Demodulation

Data bits

Viterbi Decoder

Coded bits (hard-input decoding) or Voltage Levels (soft-input decoding)
Thursday Topic: Rateless Codes

Next week’s Precepts: Lab 2