Introduction to Information Theory

Lecture 13 - Convolutional Codes & BCJR

Lecturer: *Haim Permuter* **Scribe:** *Gilad Kliger, Omri Gelbstein*

Introduction

Considering Channel Coding problem discussed in class:

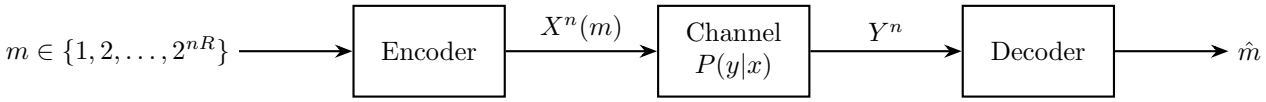

Figure 1: Standard Channel Coding Communication System.

Our goal is to reliably transmit data, which means:

$$P_e = \Pr(m \neq \hat{m}) \xrightarrow[n\to\infty]{} 0 \tag{1}$$

We already know that if the Rate is less than the channel capacity, which means $R \leq C$, then there exists a code that satisfies condition (1).

In modern communication systems, there are 4 codes that are in use:

- Convolutional Codes
- Turbo Codes
- Low-Density Parity-Check (LDPC) Codes
- Polar Codes

For LDPC and Turbo, it was demonstrated through simulations and experiments that they achieve near-capacity performance, and for Polar codes, it was proved that the code achieves the Shannon limit of achievable rate C.

As for Convolutional codes, they do not achieve the Shannon limit, but they introduced ideas such as memory, trellis diagrams, and the Viterbi algorithm that shaped some of the codes that are used today, specifically Turbo codes.

Convolutional Code

Let's consider the following encoder:

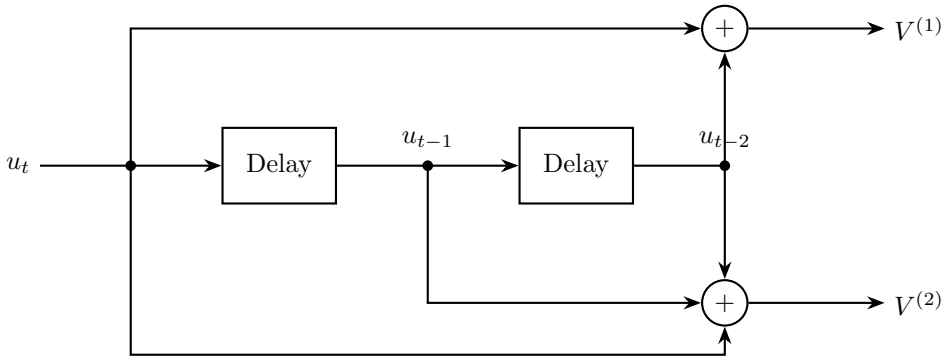

Figure 2: Basic Rate-1/2 Convolutional Code Encoder with Memory $m = 2$.

Let us define $u_t \triangleq$ Information bit, which is set to be $u_0 = 0, u_{-1} = 0$.
The entire system is generally defined as the set (n, k, m), whereas $n \triangleq$ output, $k \triangleq$ input, and $m \triangleq$ memory.
In Fig. 2, our system is defined as the set $(2, 1, 2)$, which means its rate is:

$$R = \frac{k}{n} = \frac{1}{2} \tag{2}$$

Notice that the convolution operator is defined as

$$y[n] = \sum_{k=0}^{m} x[n-k]h[k]$$

only we operate within the Binary Field. Which means the sum operator $\sum$ is replaced by the XOR operator $\oplus$, and the same goes for multiplication and the Binary AND. With respect to that let us define the system's output as

$$V^{(i)} = u * g^{(i)}$$

whereas in our example:

$$g^{(1)} = (1, 0, 1) \longrightarrow V^{(1)} = u_t \cdot g_0^{(1)} \oplus u_{t-1} \cdot g_1^{(1)} \oplus u_{t-2} \cdot g_2^{(1)} = u * g^{(1)} \tag{3}$$

$$g^{(2)} = (1, 1, 1) \longrightarrow V^{(2)} = u_t \cdot g_0^{(2)} \oplus u_{t-1} \cdot g_1^{(2)} \oplus u_{t-2} \cdot g_2^{(2)} = u * g^{(2)} \tag{4}$$

Example 1

Take the entrance vector $u = [1011100]$
By following (2) and (3) and XOR $\oplus$ properties we get that:

$$V^{(1)} = [1001011] \quad V^{(2)} = [1100101]$$

And the encoder's output will be (11 01 00 10 01 10 11) without any spaces.

State Table

Since the encoder has memory, we define its state the same way as in a FSM (Finite State Machine), to help us map out exactly how the past and present inputs combine to create the next output.

input	initial State	next State	output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	01
1	10	11	10
0	11	01	10
1	11	11	01

Trellis Diagram

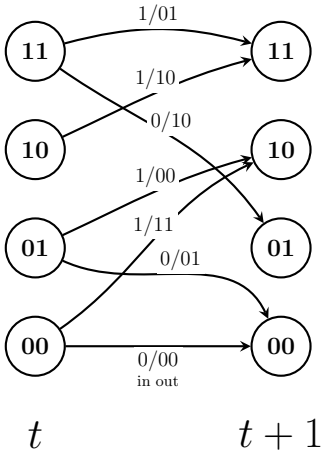

Figure 3: Single-stage State Transition Trellis Diagram.

Viterbi Algorithm

We now introduce a clever decoder known as the Viterbi algorithm, which is used for decoding convolutional codes. This algorithm was named after Andrew Viterbi, who proposed it for that purpose.

The Problem

ML - Decoder

For a memory-less channel the likelihood of the received sequence $\boldsymbol{r}$ given the transmitted codeword $\boldsymbol{v}$ is:

$$P(\boldsymbol{r}|\boldsymbol{v}) = \prod_{i=0}^{N+m-1} P(r_i|v_i) \tag{5}$$

thus the Maximum Likelihood rule defined as:

$$\hat{\boldsymbol{v}} = \arg\max_{\boldsymbol{v}} P(\boldsymbol{r}|\boldsymbol{v}) = \arg\max_{\boldsymbol{v}} \prod_{i=0}^{N+m-1} P(r_i|v_i) \tag{6}$$

Taking advantage of the monotonic property of the log function we can present the ML rule as:

$$\hat{\boldsymbol{v}} = \arg\max_{\boldsymbol{v}} \log(P(\boldsymbol{r}|\boldsymbol{v})) = \arg\max_{\boldsymbol{v}} \sum_{i=0}^{N+m-1} \log P(r_i|v_i) \tag{7}$$

Considering a Binary Symmetric Channel (BSC) with crossover probability p:

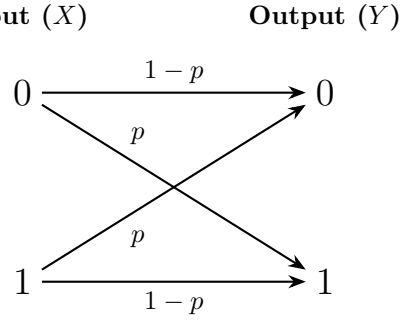

Figure 4: Binary Symmetric Channel (BSC) Probability Transition Model.

The problem is equivalent to minimizing the Hamming distance between the received sequence and the chosen codeword:

$$\hat{\boldsymbol{v}} = \arg\min_{\boldsymbol{v}} d_H(\boldsymbol{r}, \boldsymbol{v}) \tag{8}$$

when the Hamming distance is defined as:

$$d_H(\boldsymbol{r}, \boldsymbol{v}) = \sum_i \mathbf{1}\{r_i \neq v_i\} \tag{9}$$

Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence. For instance, when $N = 256$ bits, the number of possibilities rivals the number of atoms in the universe!

Solution - Viterbi Decoder

The Viterbi decoding algorithm uses two metrics:

- Branch Metric (BM)
- Path Metric (PM)

The branch metric is the "cost" between what was transmitted and what was received. In our case of BSC and hard decision decoding, the branch metric is the *Hamming distance* between the expected bits and the received ones and is defined for each arrow in the trellis.

The path metric is a value associated with each state in the trellis. In our case, it represents the Hamming distance of the most likely path from the initial state to the current state in the trellis. Again, by "most likely", we mean the path with the minimum Hamming distance between the states.

We can compute the path metric at each time and state using the path metrics of previously computed states and branch metrics.

Path Metric Computation

We denote the path metric of state s at time i as: $\text{PM}[s, i]$.

Observe that for each state s at time $i + 1$, there are only two possible states at time i, which we denote as $\alpha and \beta$.

Now, to choose the "surviving path" and minimize the cost, we present the decision rule for each step:

$$\text{PM}[s, i+1] = \min\left(\text{PM}[\alpha, i] + \text{BM}[\alpha \longrightarrow s], \text{PM}[\beta, i] + \text{BM}[\beta \longrightarrow s]\right) \tag{10}$$

Most likely Path

We begin by associating the state $'00'$ with a cost of 0 and all others with a cost of ∞. The main loop consists of two steps:

- Calculating the branch metric for each arc in the trellis.
- Computing the path metric.

where the path metric computation consists of three steps:

- Add the branch metric to the path metric of the previous state.
- Compare the sum of both paths arriving at the state.
- Select the path with the smallest path metric value.

At the end we are left with only one surviving path.

Observe that the complexity of the Viterbi algorithm is $\mathcal{O}(N)$ instead of $\mathcal{O}(2^N)$.
The key insight is that we are relying on "future knowledge" in the decoding process, and that is causing a delay. For that reason, we use a "rule of thumb" of decoding sequences of $5K$ length, where K is the memory size.

Example 2

Let us define $'u'$ as the original message we sent into our encoder, $'v'$ is the encoded message going into the noisy BSC channel, and finally $'r'$ is the received sequence at the other side. Suppose that the received sequence is $r = (01\ 11\ 10\ 10\ 00\ 11\ 10)$, our goal is to figure out which encoded message $'v'$ was most likely sent? As we previously explained we use the Viterbi Algorithm, and operate under the assumption that the initial state was 00.

Step t=1 | Received 01

We only have two possible branches starting from the initial state 00:

- **Move to state 00:** Expected output is 00.
  $\text{distance}(00, 01) = 1 \longrightarrow$ Total cost $= 1$.
- **Move to state 10:** Expected output is 11.
  $\text{distance}(11, 01) = 1 \longrightarrow$ Total cost $= 1$.

Step t=2 | Received 11

Now we branch out from the two states we reached in Step 1 (states 00 and 10):

- **From state 00** (Current cost: 1):
  - Move to 00 (expected 00): $1 + \text{distance}(00, 11) = 1 + 2 = 3$.
  - Move to 10 (expected 11): $1 + \text{distance}(11, 11) = 1 + 0 = 1$.
- **From state 10** (Current cost: 1):
  - Move to 01 (expected 01): $1 + \text{distance}(01, 11) = 1 + 1 = 2$.
  - Move to 11 (expected 10): $1 + \text{distance}(10, 11) = 1 + 1 = 2$.

Step t=3 | Received 10

Now we branch out from the four states we reached in Step 2:

- **Paths going into state 11:**
  - From 10 (cost 1): Expected 10. $1 + \text{distance}(10, 10) = 1 + 0 = 1$.
  - From 11 (cost 2): Expected 01. $2 + \text{distance}(01, 10) = 2 + 2 = 4$.
  - **Decision:** $1 < 4$, we keep the path from state 10.
- **Paths going into state 10:**
  - From 00 (cost 3): Expected 11. $3 + \text{distance}(11, 10) = 3 + 1 = 4$.
  - From 01 (cost 2): Expected 00. $2 + \text{distance}(00, 10) = 2 + 1 = 3$.
  - **Decision:** $3 < 4$, we keep the path from state 01.
- **Paths going into state 01:**
  - From 10 (cost 1): Expected 01. $1 + \text{distance}(01, 10) = 1 + 2 = 3$.
  - From 11 (cost 2): Expected 10. $2 + \text{distance}(10, 10) = 2 + 0 = 2$.
  - **Decision:** $2 < 3$, we keep the path from state 11.
- **Paths going into state 00:**
  - From 00 (cost 3): Expected 00. $3 + \text{distance}(00, 10) = 3 + 1 = 4$.
  - From 01 (cost 2): Expected 11. $2 + \text{distance}(11, 10) = 2 + 1 = 3$.
  - **Decision:** $3 < 4$, we keep the path from state 01.

Final Step

We continue the same way for the rest of the steps and finish the Trellis diagram. Notice in Fig. 5 the Trellis diagram shrinks back into 1 state at t=7, that happens because we add zeros at the end of the original message $'u'$ to reset the memory.
By the time we are through with the diagram, we mark the 'cheapest' path and start tracing back. Each branch has the (input/output) mark, by tracing back on the cheapest path (the yellow path in Fig. 5) each input is part of the original message $'u'$, and each output is part of the encoded message $'v'$.

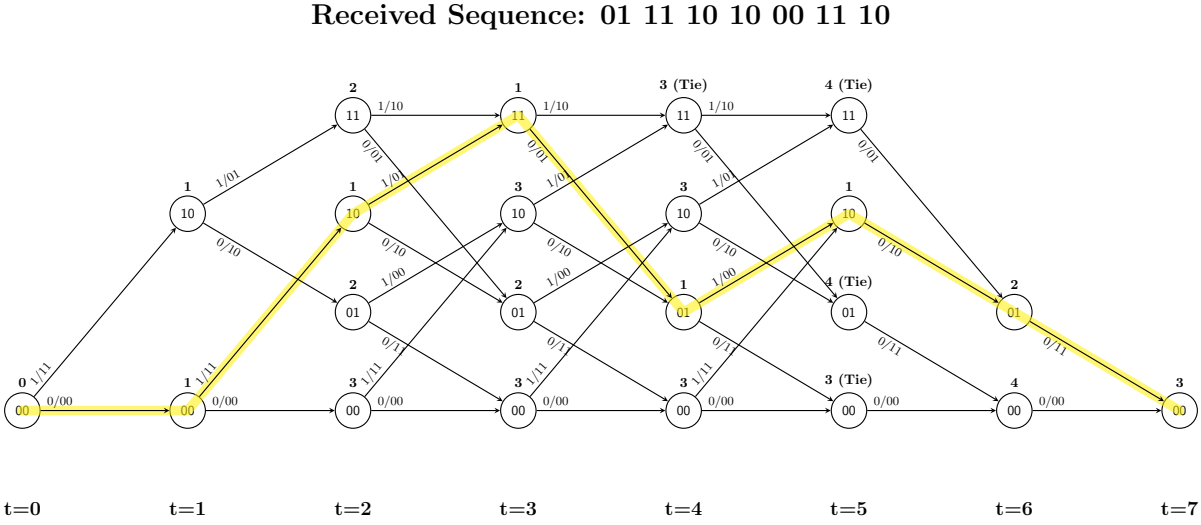

Figure 5: Complete Viterbi Decoding Trellis Graph for Example 2 (with Backtracking Pathway Highlighted).

BCJR Algorithm

The problem with the Viterbi Algorithm

When we talked about the Viterbi algorithm our goal was to find the encoded message v that was most likely sent. This is helpful when you need the entire message to be correct as a whole. However the algorithm has two main flaws:

1. Viterbi only cares about the one winning path, and throws away all the other paths. Sometimes, those losing paths actually have the correct bit. By ignoring them, the algorithm can make a mistakes on a single bit.

2. It does not tell you how sure it is. Viterbi only gives a hard answer: 0 or 1. It cannot tell us how sure it is, and modern systems need to know how sure the guess is so they can fix errors better.

The solution - BCJR Algorithm

Instead of looking for one winning path and throwing the rest away (like Viterbi does), let us use the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm, which keeps and uses all possible paths. Its main goal is to protect each bit individually. It calculates the exact probability of each specific bit being a '1' or a '0', rather than just guessing the whole message at once.
Let us define u_l as the specific bit in the original message at $t = l$, and r the same as before as the whole received message at the end of the channel.

Log Likelihood Ratio

Our goal is to minimize the bit error rate (BER) by using the following decision rule:

$$\hat{u}_l = \arg\max_{\hat{u}_l} P(u_l = \hat{u}_l \mid \mathbf{r}) \tag{11}$$

when u_l is the transmitted bit and $\mathbf{r}$ is the received signal assuming BPSK modulation we present the log likelihood ratio (LLR):

$$L(u_l) = \ln\left[\frac{P(u_l = +1 \mid \mathbf{r})}{P(u_l = -1 \mid \mathbf{r})}\right] \tag{12}$$

We can rewrite the numinator as:

$$P(u_l = +1 \mid \mathbf{r}) = \frac{P(u_l = +1, \mathbf{r})}{P(\mathbf{r})} \tag{13}$$

$$= \frac{\sum_{u \in u_l^+} P(\mathbf{r} \mid v(u)) \cdot P(u)}{\sum_{u \in u} P(\mathbf{r} \mid v(u)) \cdot P(u)} \tag{14}$$

when we get (13) from Bayes and u_l^+ is the set of all information sequences s.t $u_l = +1$.
So we can define the LLR as:

$$L(u_l) = \ln\left[\frac{\sum_{u \in u_l^+} P(\mathbf{r} \mid v(u)) \cdot P(u)}{\sum_{u \in u_l^-} P(\mathbf{r} \mid v(u)) \cdot P(u)}\right] \tag{15}$$

$$= \ln\left[\frac{\sum_{s', s \in \Sigma_l^+} P(s_l = s', s_{l+1} = s, \mathbf{r})}{\sum_{s', s \in \Sigma_l^-} P(s_l = s', s_{l+1} = s, \mathbf{r})}\right] \tag{16}$$

Note that v is a deterministic function of the input u:

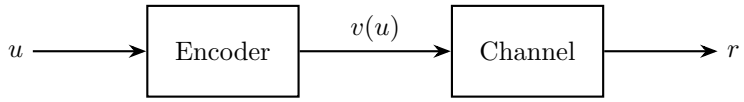

Figure 6: Deterministic Relation between Encoder Input and Channel Input.

in (16) Σ_l^+ represents all pairs of states (s', s) in relation to $u_l = +1$

The BCJR algorithm is a clever and efficient way to calculate the probabilities in (16)

$$\begin{aligned} P(s', s', \mathbf{r}) &= P(s', s, \mathbf{r}_{t<l}, \mathbf{r}_l, \mathbf{r}_{t>l}) \\ &= P(\mathbf{r}_{t>l} \mid s', s, \mathbf{r}_l, \mathbf{r}_{t<l}) \cdot P(s', s, \mathbf{r}_{t<l}, \mathbf{r}_l) \\ &= P(\mathbf{r}_{t>l} \mid s) \cdot P(s', s, \mathbf{r}_l, \mathbf{r}_{t<l}) \\ &= P(\mathbf{r}_{t>l} \mid s) \cdot P(s, \mathbf{r}_l \mid s', \mathbf{r}_{t<l}) \cdot P(s', \mathbf{r}_{t<l}) \\ &= P(\mathbf{r}_{t>l} \mid s) \cdot P(s, \mathbf{r}_l \mid s') \cdot P(s', \mathbf{r}_{t<l}) \end{aligned} \tag{17}$$

The Algorithm

First, we denote the probabilities in (17) as:

$$\begin{aligned} \alpha_l(s') &\triangleq P(s', \mathbf{r}_{t<l}) \\ \gamma_l(s', s) &\triangleq P(s, \mathbf{r}_l \mid s') \\ \beta_l(s) &\triangleq P(\mathbf{r}_{t>l} \mid s) \end{aligned}$$

Note that:

$$\alpha_l(s') \cdot \beta_l(s) \cdot \gamma_l(s', s) = P(s', s', \mathbf{r}) \tag{18}$$

α and β calculation: We can calculate α and β by using their previous and next values.
For α:

$$\begin{aligned} \alpha_{l+1}(s) = P(s, \mathbf{r}_{t<l+1}) &= \sum_{s'} P(s', s, \mathbf{r}_{t<l+1}) = \sum_{s'} P(s', s, \mathbf{r}_l, \mathbf{r}_{t<l}) \\ &= \sum_{s'} P(\mathbf{r}_l, s \mid s', \mathbf{r}_{t<l}) \cdot P(s', \mathbf{r}_{t<l}) = \sum_{s'} \gamma_l(s', s) \alpha_l(s') \end{aligned} \tag{19}$$

The forward recursion is initialized at time $t = 0$ with the following boundary state probabilities:

$$\alpha_0(s') = \begin{cases} 1 & s' = 0 \\ 0 & s' \neq 0 \end{cases} \tag{20}$$

As for β we note that:

$$\begin{aligned} \beta_l(s') &= P(\mathbf{r}_{t>l-1} \mid s') = \sum_s P(\mathbf{r}_{t>l-1}, s \mid s') \\ &= \sum_s P(\mathbf{r}_{t>l}, \mathbf{r}_l, s \mid s') \\ &= \sum_s P(\mathbf{r}_l, s \mid s') \cdot P(\mathbf{r}_{t>l} \mid s', s, \mathbf{r}_l) \\ &= \sum_s \gamma_l(s', s)\beta_{l+1}(s) \end{aligned} \tag{21}$$

Assuming the encoder always terminates at the all-zero state at the final stage $t = K$, the recursion is initialized as:

$$\beta_K(s) = \begin{cases} 1 & s = 0 \\ 0 & s \neq 0 \end{cases} \tag{22}$$

So the values can be computed by forward recursion for α and backward for β.

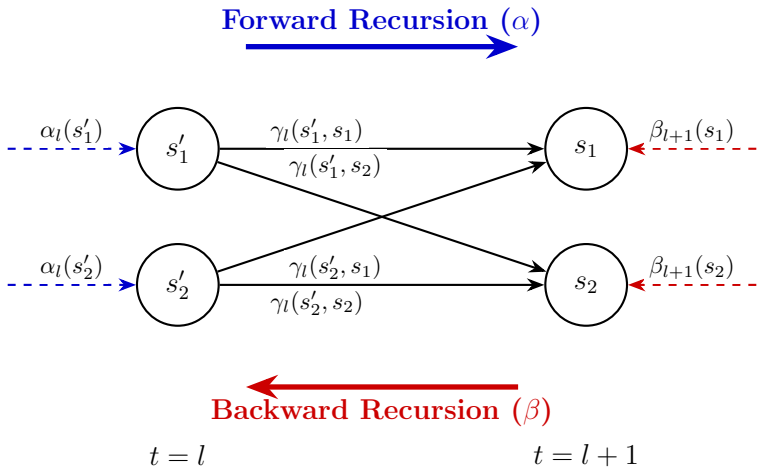

Figure 7: BCJR Recursions Flow: α propagates forward in time, while β propagates backward in time.

γ calculation: γ is calculated for each arc in the trellis according to the received signal $\mathbf{r}_l$

$$\gamma_l(s', s) = P(s, \mathbf{r}_l \mid s') = P(s \mid s') \cdot P(\mathbf{r}_l \mid s', s) = P(u(s', s)) \cdot P(\mathbf{r}_l \mid v(s', s)) \tag{23}$$

For an AWGN channel with noise power spectral density $N_0/2$, the channel conditional probability is Gaussian, and the metric becomes:

$$\gamma_l(s', s) = P(u) \cdot \left(\frac{1}{\pi N_0}\right)^{n/2} e^{-\frac{1}{N_0}\|\mathbf{r}_l - \sqrt{E_s}v(s', s)\|^2} \tag{24}$$

Step by Step

Step 1: Initialization Initiate the forward and backward boundary conditions:

$$\alpha_0(s') = \begin{cases} 1 & s' = 0 \\ 0 & s' \neq 0 \end{cases}$$

$$\beta_K(s) = \begin{cases} 1 & s = 0 \\ 0 & s \neq 0 \end{cases}$$

Step 2: γ Calculation compute:

$$\gamma_l(s', s) \ , l = 1, \ldots, K \ , s', s \in S$$

Step 3: Forward and Backward Recursion compute:

$$\alpha_{l+1}(s) \ , l = 0, 1, \ldots, K-1$$
$$\beta_l(s) \ , l = 1, \ldots, K$$

Step 4: Final Computation compute $L(u_l)$ using α , β and γ

Summary of BCJR

Unlike the Viterbi algorithm, which only looks for one "winning" path and throws all the rest away, the BCJR algorithm keeps and uses all possible paths in the trellis. Instead of guessing the whole message at once (hard decision), it calculates the exact probability for each individual bit being a '1' or a '0' (soft output).
This soft output is a huge advantage when working with multiple decoders. We can take the probability we calculated for a bit and pass it to the next decoder as a bias ,replacing the initial $P(u)$ in equation (24) to help it be more accurate. (In fact, this continuous teamwork where decoders pass information back and forth is exactly how **Turbo Decoding** works.)