Sequence to Sequence Modeling

- Problem:
  - Input Sequence: $X_1 \ldots X_n$
  - Output Sequence: $Y_1 \ldots Y_m$

- $|X| \neq |Y|$
In HW1P2, we utilized sequence classification for phoneme recognition. We can manage this problem by applying a variant using recurrent nets.

Left: Sequence of inputs produces a single output; Right: How???
New: Complex Problem - Training

- Objective: Given a sequence of inputs, asynchronously output a sequence of symbols
  - Concatenation of many copies of the simple model in the previous slide
In the previous model, we ignored intermediate steps. However, we can exploit the untagged inputs and assume the same output.

How do we know when to output symbols?

- Apply our ideas from HW1P2:
  - At each time in the network outputs a probability for each output symbol given all inputs until that time.
  - The most likely symbol sequence given the inputs. How?
Possible Solutions

- **Solution 1**: Simply select the most probable symbol at each time. Merge adjacent repeated symbols, and place the actual emission of the symbol in the final instant.
  - **Issue 1**: This isn’t the most probable sequence of symbols
  - **Issue 2**: Cannot distinguish between an extended symbol and repetitions of the symbol

- **Solution 2**: Impose external constraints on what sequences are allowed
  - **Issue 1**: A suboptimal decode that actually finds the most likely time-synchronous output sequence. Will be discussed in lecture.

Lecture will discuss computing Divergence
Overall Solution:

- Apply both previous solutions
  - At each time the network outputs a probability for each output symbol
- Block out all rows that do not include symbols from the target sequence
- Compose a graph such that every path in the graph from source to sink represents a valid alignment
Find the most probable sequence of symbols using the graph above

- Edge scores have a probability of 1
- Nodes scores are probabilities resulting from the neural network

Lecture will discuss how to find the most probable sequence given the graph and how to compute the divergence once we get the most probable sequence.
Repetition Issue and Solution

- We have a decode:
  - R R R O O O O D
  - Is this the symbol sequence ROD or ROOD?

- Introduce an explicit extra symbol which serves to separate discrete versions of a symbol (Blank)
  - RRR---OO---DDD = ROD
  - –RR-R---OO---D-DD = RRODD

- The label recognized by the network must now include the extra blank symbol that will need to be trained
Final Graph
CTC - Training Procedure

1. Setup Network
   a. Many LSTM
2. Initialize network with a Blank Symbol
3. Pass training instances through network to obtain probabilities for all labels/symbols
4. Construct graph on previous page
5. Forward and Backward Algorithm - Lecture
6. Compute Divergence - Lecture
7. Update Parameters

Connectionist Temporal Classification
How to decode at test time?

- I will first discuss an example of training a network using nn.CTCLoss

- Then Tony will discuss an algorithm called Beam Search using pseudocode and an example
The forward output

<table>
<thead>
<tr>
<th>/AH/</th>
<th>y₀^b</th>
<th>y₁^b</th>
<th>y₂^b</th>
<th>y₃^b</th>
<th>y₄^b</th>
<th>y₅^b</th>
<th>y₆^b</th>
<th>y₇^b</th>
<th>y₈^b</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B/</td>
<td>y₀^b</td>
<td>y₁^b</td>
<td>y₂^b</td>
<td>y₃^b</td>
<td>y₄^b</td>
<td>y₅^b</td>
<td>y₆^b</td>
<td>y₇^b</td>
<td>y₈^b</td>
</tr>
<tr>
<td>/D/</td>
<td>y₀^D</td>
<td>y₁^D</td>
<td>y₂^D</td>
<td>y₃^D</td>
<td>y₄^D</td>
<td>y₅^D</td>
<td>y₆^D</td>
<td>y₇^D</td>
<td>y₈^D</td>
</tr>
<tr>
<td>/EH/</td>
<td>y₀^EH</td>
<td>y₁^EH</td>
<td>y₂^EH</td>
<td>y₃^EH</td>
<td>y₄^EH</td>
<td>y₅^EH</td>
<td>y₆^EH</td>
<td>y₇^EH</td>
<td>y₈^EH</td>
</tr>
<tr>
<td>/IY/</td>
<td>y₀^IY</td>
<td>y₁^IY</td>
<td>y₂^IY</td>
<td>y₃^IY</td>
<td>y₄^IY</td>
<td>y₅^IY</td>
<td>y₆^IY</td>
<td>y₇^IY</td>
<td>y₈^IY</td>
</tr>
<tr>
<td>/F/</td>
<td>y₀^F</td>
<td>y₁^F</td>
<td>y₂^F</td>
<td>y₃^F</td>
<td>y₄^F</td>
<td>y₅^F</td>
<td>y₆^F</td>
<td>y₇^F</td>
<td>y₈^F</td>
</tr>
<tr>
<td>/G/</td>
<td>y₀^G</td>
<td>y₁^G</td>
<td>y₂^G</td>
<td>y₃^G</td>
<td>y₄^G</td>
<td>y₅^G</td>
<td>y₆^G</td>
<td>y₇^G</td>
<td>y₈^G</td>
</tr>
</tbody>
</table>

Diagram showing the forward output with inputs X₀ to X₈.
Returning to the decoding problem

How to decode at test time?

- Greedy decode -> choose symbol with highest probability at each time step and merge
  - Sub-optimal decode which finds most likely synchronous output sequence

- Objective of decoding -> Most likely asynchronous symbol sequence
  - Find all decodings and pick the most likely decode!
  - Unfortunately, explicit computation of this will require evaluation of an exponential number of symbol sequences
  - Solution: Organize all possible symbol sequences as a (semi)tree
Hypothesis semi-tree

- The semi tree of hypotheses (assuming only 3 symbols in the vocabulary)
- Every symbol connects to every symbol other than itself
- It also connects to a blank, which connects to every symbol including itself
- The simple structure repeats recursively
- Each node represents a unique symbol sequence!
Decoding graph for the tree

- The figure to the left is the tree, drawn in a vertical line.
- The graph is just the tree unrolled over time.
- The alpha at final time represents the full forward score for a unique symbol sequence.
- Select the symbol sequence with the largest alpha.
Pruning

- This is the “theoretically correct” CTC decoder
- In practice, the graph gets exponentially large very quickly
- To prevent this pruning strategies are employed to keep the graph (and computation) manageable

Beam Search

Inputs:
- BeamWidth: int that is the number of paths considered
- SymbolSet: set of symbols, not including blank
- $y$: array of probabilities of shape $(\text{len(SymbolSet)} + 1, t)$
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, # including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])

end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewPathScore, BlankPathScore, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])
end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score
function InitializePaths(SymbolSet, y)

InitialBlankPathScore = [], InitialPathScore = []
# First push the blank into a path-ending-with-blank stack. No symbol has been invoked yet
path = null
InitialBlankPathScore[path] = y[blank]  # Score of blank at t=1
InitialPathsWithFinalBlank = {path}

# Push rest of the symbols into a path-ending-with-symbol stack
InitialPathsWithFinalSymbol = {}
for c in SymbolSet  # This is the entire symbol set, without the blank
    path = c
    InitialPathScore[path] = y[c]  # Score of symbol c at t=1
    InitialPathsWithFinalSymbol += path  # Set addition
end

return InitialPathsWithFinalBlank, InitialPathsWithFinalSymbol, InitialBlankPathScore, InitialPathScore
**BEAM SEARCH**

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, # including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T

  # Prune the collection down to the BeamWidth
  PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

  # First extend paths by a blank
  NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

  # Next extend paths by a symbol
  NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])

end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score

We will visit this routine after discussing the rest of the loop (to avoid confusion)
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)
    
    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])
    
    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])
end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score
Only transitions into nodes on the rows corresponding to blanks.
**BEAM SEARCH: Extending with blanks**

Global PathScore, BlankPathScore

function **ExtendWithBlank** (PathsWithTerminalBlank, PathsWithTerminalSymbol, y)

    UpdatedPathsWithTerminalBlank = {}
    UpdatedBlankPathScore = []

    # First work on paths with terminal blanks
    #(This represents transitions along horizontal trellis edges for blanks)
    for path in PathsWithTerminalBlank:
        # Repeating a blank doesn't change the symbol sequence
        UpdatedPathsWithTerminalBlank += path  # Set addition
        UpdatedBlankPathScore[path] = BlankPathScore[path] * y[blank]

    # Then extend paths with terminal symbols by blanks
    for path in PathsWithTerminalSymbol:
        # If there is already an equivalent string in UpdatedPathsWithTerminalBlank
        # simply add the score. If not create a new entry
        if path in UpdatedPathsWithTerminalBlank
            UpdatedBlankPathScore[path] += PathScore[path] * y[blank]
        else
            UpdatedPathsWithTerminalBlank += path  # Set addition
            UpdatedBlankPathScore[path] = PathScore[path] * y[blank]

    return UpdatedPathsWithTerminalBlank, UpdatedBlankPathScore
BEAM SEARCH: Extending with blanks

Global PathScore, BlankPathScore

function ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y)
  UpdatedPathsWithTerminalBlank = {}
  UpdatedBlankPathScore = []

  # First work on paths with terminal blanks
  # (This represents transitions along horizontal trellis edges for blanks)
  for path in PathsWithTerminalBlank:
    # Repeating a blank doesn’t change the symbol sequence
    UpdatedPathsWithTerminalBlank += path  # Set addition
    UpdatedBlankPathScore[path] = BlankPathScore[path] * y[blank]

  # Then extend paths with terminal symbols by blanks
  for path in PathsWithTerminalSymbol:
    # If there is already an equivalent string in UpdatedPathsWithTerminalBlank
    # simply add the score. If not create a new entry
    if path in UpdatedPathsWithTerminalBlank
      UpdatedBlankPathScore[path] += PathScore[path] * y[blank]
    else
      UpdatedPathsWithTerminalBlank += path  # Set addition
      UpdatedBlankPathScore[path] = PathScore[path] * y[blank]

  return UpdatedPathsWithTerminalBlank, UpdatedBlankPathScore
Transitions from “blank” lines to “blank” lines (which will all be horizontal edges)
BEAM SEARCH: Extending with blanks

Global PathScore, BlankPathScore

function ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y)
    UpdatedPathsWithTerminalBlank = {}
    UpdatedBlankPathScore = []
    # First work on paths with terminal blanks
    # (This represents transitions along horizontal trellis edges for blanks)
    for path in PathsWithTerminalBlank:
        # Repeating a blank doesn’t change the symbol sequence
        UpdatedPathsWithTerminalBlank += path  # Set addition
        UpdatedBlankPathScore[path] = BlankPathScore[path] * y[blank]
    end

    # Then extend paths with terminal symbols by blanks
    for path in PathsWithTerminalSymbol:
        # If there is already an equivalent string in UpdatedPathsWithTerminalBlank
        # simply add the score. If not create a new entry
        if path in UpdatedPathsWithTerminalBlank
            UpdatedBlankPathScore[path] += PathScore[path] * y[blank]
        else
            UpdatedPathsWithTerminalBlank += path  # Set addition
            UpdatedBlankPathScore[path] = PathScore[path] * y[blank]
        end
    end

    return UpdatedPathsWithTerminalBlank,
          UpdatedBlankPathScore
Transitions from “symbol” lines to “blank” lines
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, 
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = 
InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, SymbolSet, y[:,t])

end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, 
NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score
Only transitions into nodes on the rows corresponding to non-blank symbols

(figure shows path extensions for only 2 time steps)
Global PathScore, BlankPathScore

function ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y)
    UpdatedPathsWithTerminalSymbol = {}
    UpdatedPathScore = []

    # First extend the paths terminating in blanks. This will always create a new sequence
    for path in PathsWithTerminalBlank:
        for c in SymbolSet:  # SymbolSet does not include blanks
            newpath = path + c  # Concatenation
            UpdatedPathsWithTerminalSymbol += newpath  # Set addition
            UpdatedPathScore[newpath] = BlankPathScore[path] * y(c)
    
    # Next work on paths with terminal symbols
    for path in PathsWithTerminalSymbol:
        # Extend the path with every symbol other than blank
        for c in SymbolSet:  # SymbolSet does not include blanks
            newpath = (c == path[end]) ? path : path + c  # Horizontal transitions don’t extend the sequence
            if newpath in UpdatedPathsWithTerminalSymbol:  # Already in list, merge paths
                UpdatedPathScore[newpath] += PathScore[path] * y[c]
            else:  # Create new path
                UpdatedPathsWithTerminalSymbol += newpath  # Set addition
                UpdatedPathScore[newpath] = PathScore[path] * y[c]
    
    return UpdatedPathsWithTerminalSymbol, UpdatedPathScore
BEAM SEARCH: Extending with symbols

Global PathScore, BlankPathScore

function ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y)
    UpdatedPathsWithTerminalSymbol = {}
    UpdatedPathScore = []

    # First extend the paths terminating in blanks. This will always create a new sequence
    for path in PathsWithTerminalBlank:
        for c in SymbolSet:
            # SymbolSet does not include blanks
            newpath = path + c  # Concatenation
            UpdatedPathsWithTerminalSymbol += newpath  # Set addition
            UpdatedPathScore[newpath] = BlankPathScore[path] * y(c)
    
    # Next work on paths with terminal symbols
    for path in PathsWithTerminalSymbol:
        # Extend the path with every symbol other than blank
        for c in SymbolSet:
            # SymbolSet does not include blanks
            newpath = (c == path[end]) ? path : path + c  # Horizontal transitions don’t extend the sequence
            if newpath in UpdatedPathsWithTerminalSymbol:  # Already in list, merge paths
                UpdatedPathScore[newpath] += PathScore[path] * y[c]
            else:  # Create new path
                UpdatedPathsWithTerminalSymbol += newpath  # Set addition
                UpdatedPathScore[newpath] = PathScore[path] * y[c]
        
    return UpdatedPathsWithTerminalSymbol, UpdatedPathScore

( only at t=1)
UpdatedPathsWithTerminalSymbol

}\nS1
S2
Transitions from “blank” lines to “symbol” lines

(figure shows path extensions for only 2 time steps)
BEAM SEARCH: Extending with symbols

Global PathScore, BlankPathScore

function ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y)
    UpdatedPathsWithTerminalSymbol = {}
    UpdatedPathScore = []

    # First extend the paths terminating in blanks. This will always create a new sequence
    for path in PathsWithTerminalBlank:
        for c in SymbolSet:
            # SymbolSet does not include blanks
            newpath = path + c  # Concatenation
            UpdatedPathsWithTerminalSymbol += newpath  # Set addition
            UpdatedPathScore[newpath] = BlankPathScore[path] * y(c)
        end
    end

    # Next work on paths with terminal symbols
    for path in PathsWithTerminalSymbol:
        # Extend the path with every symbol other than blank
        for c in SymbolSet:
            # SymbolSet does not include blanks
            newpath = (c == path[end]) ? path : path + c  # Horizontal transitions don’t extend the sequence
            if newpath in UpdatedPathsWithTerminalSymbol:  # Already in list, merge paths
                UpdatedPathScore[newpath] += PathScore[path] * y[c]
            else  # Create new path
                UpdatedPathsWithTerminalSymbol += newpath  # Set addition
                UpdatedPathScore[newpath] = PathScore[path] * y[c]
            end
        end
    end

return UpdatedPathsWithTerminalSymbol, UpdatedPathScore
Transitions from “symbol” lines to “symbol” lines (including horizontal transitions)

(figure shows path extensions for only 2 time steps)
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, # including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, PathScore, BlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])

end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score
Consider this instant
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end
    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist) # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end
    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end
    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end

    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p  # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end

    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p  # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end

    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore
Consider this instant

Aggregate scores from both “symbol” rows and “blank” rows
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end
    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p  # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end
    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p  # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end
    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end
    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p  # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end
    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p  # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end
    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore
Consider this instant

Effectively, prune out nodes on “blank” rows with scores below cutoff

They will subsequently not contribute to the computation

Retain nodes on “blank” rows with scores above cutoff

\[ X_0, X_1, X_2, X_3, X_4 \]
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end
    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end
    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end
    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore

Find nodes on “symbol” rows with scores above cutoff and add them to the “active” list
Consider this instant

Effectively prune out nodes on “symbol” rows with scores below cutoff

They will subsequently not contribute to the computation

Retain nodes on “symbol” rows with scores above cutoff

Effectively prune out nodes on “symbol” rows with scores below cutoff
Effectively prune out nodes on “symbol” rows with scores below cutoff. They will subsequently not contribute to the computation.

Consider this instant:

Retain nodes on “symbol” rows with scores above cutoff.
BEAM SEARCH: Pruning low-scoring entries

Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end
    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ? scorelist[BeamWidth] : scorelist[end]
    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p  # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end
    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p  # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end
    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore

The overall effect of these steps:
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
  # Prune the collection down to the BeamWidth
  PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

  # First extend paths by a blank
  NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

  # Next extend paths by a symbol
  NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])
end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore) # Find the path with the best score

Why is the pruning here and not at the end of the loop?

Because we don't want to prune paths at the final time. This loses information.
Instead at the final time we will merge paths that represent the same symbol sequence.
BEAM SEARCH

Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, # including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])
end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore) # Find the path with the best score
Merge scores for "S_2" and "S_2-"

Merge scores for "S_1" and "S_1-"
BEAM SEARCH: Merging final paths

Global PathScore, BlankPathScore

function MergeIdenticalPaths(PathsWithTerminalBlank, PathsWithTerminalSymbol)

    # All paths with terminal symbols will remain
    MergedPaths = PathsWithTerminalSymbol
    FinalPathScore = PathScore

    # Paths with terminal blanks will contribute scores to existing identical paths from
    # PathsWithTerminalSymbol if present, or be included in the final set, otherwise
    for p in PathsWithTerminalBlank
        if p in MergedPaths
            FinalPathScore[p] += BlankPathScore[p]
        else
            MergedPaths += p  # Set addition
            FinalPathScore[p] = BlankPathScore[p]
        end
    end

return MergedPaths, FinalPathScore
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols, including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore = InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, PathScore, BlankPathScore = Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore, BeamWidth)

    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y[:,t])
end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore, NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore)  # Find the path with the best score