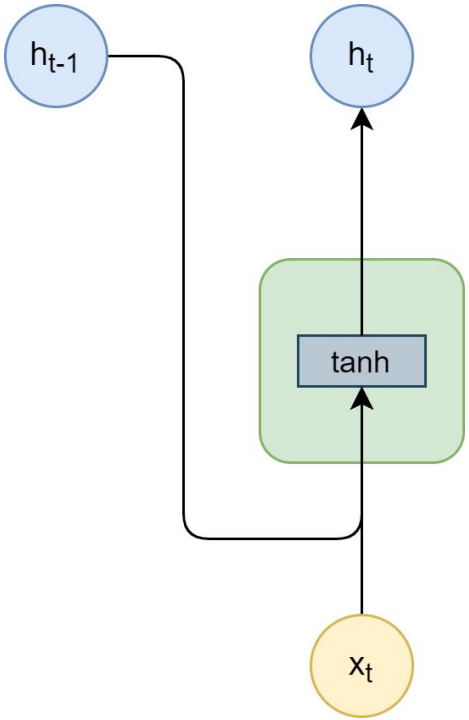
HW3P1

RNN, GRU, CTC, and Greedy / Beam Search

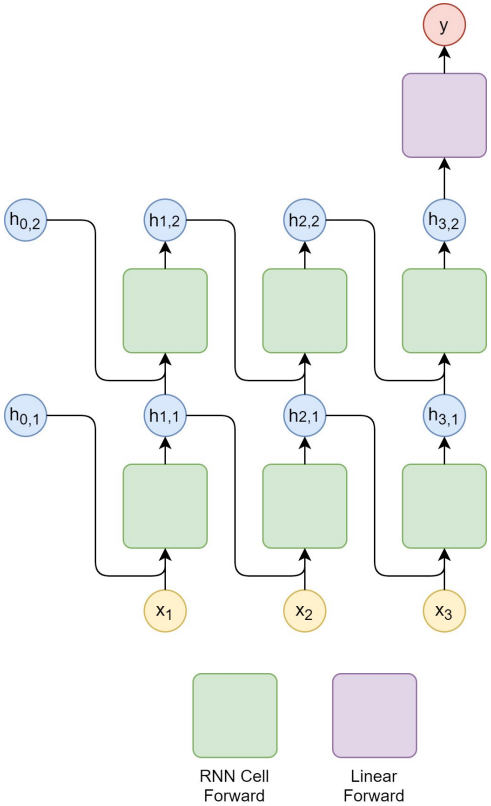
Yuxin(Abbey) Pei, Zilin Si
Oct 27, 2021

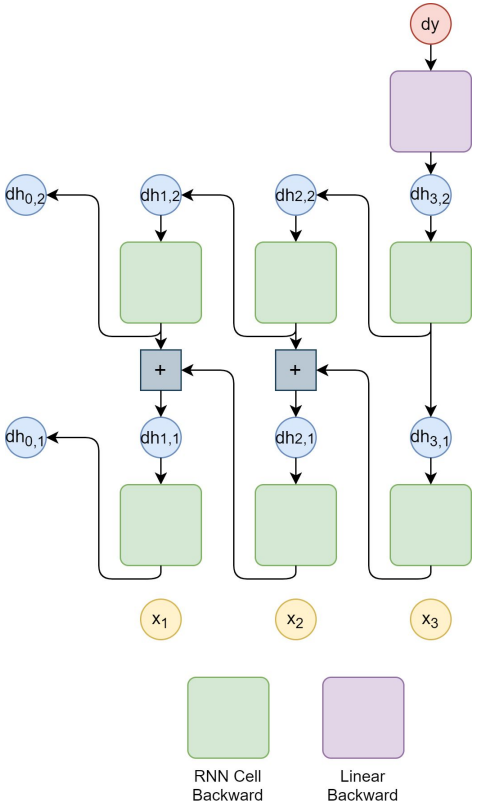
RNN Cell Forward / Backward



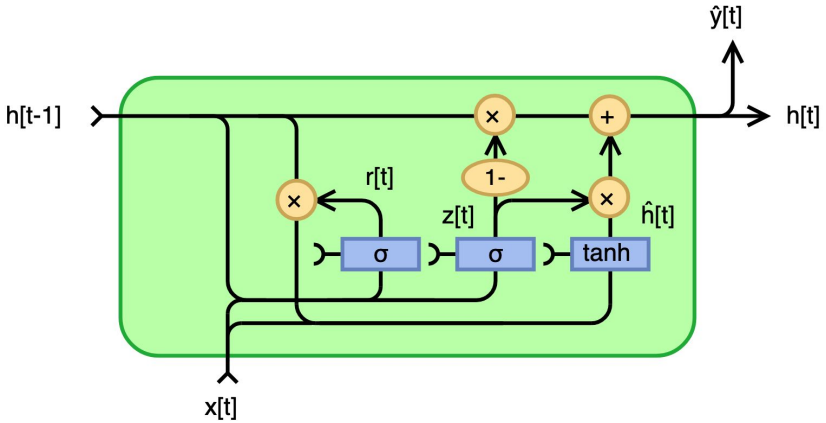
$$h_t = tanh(W_{ih} x_t + b_{ih} + W_{hh} h_{t-1} + b_{hh})$$

RNN Phoneme Classifier





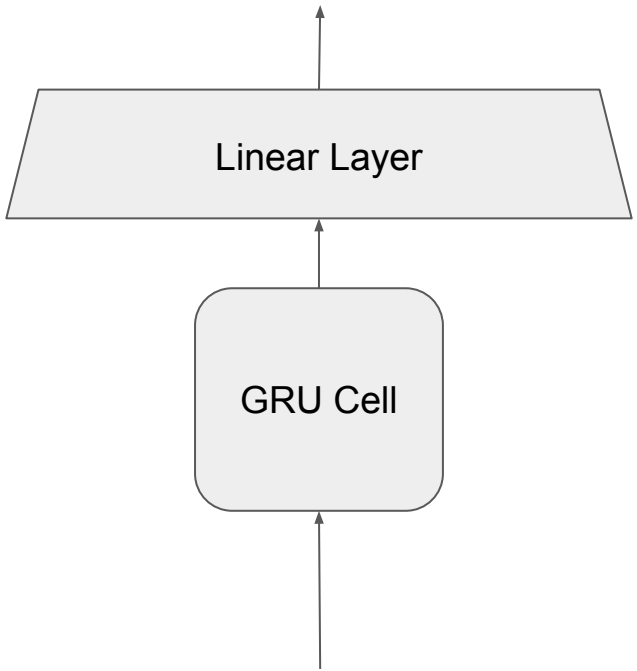
GRU Cell Forward / Backward



$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr})$$
$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz})$$
$$\mathbf{n}_t = \tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \otimes (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn}))$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \otimes \mathbf{n}_t + \mathbf{z}_t \otimes \mathbf{h}_{t-1}$$

https://colah.github.io/posts/2015-08-Backprop

GRU Inference



CTC - mytorch/ctc.py

```
def targetWithBlank(self, target):
    """Extend target sequence with blank.

def forwardProb(self, logits, extSymbols, skipConnect):
    """Compute forward probabilities.

def backwardProb(self, logits, extSymbols, skipConnect):
    """Compute backward probabilities.

def postProb(self, alpha, beta):
    """Compute posterior probabilities.
```

FORWARD ALGORITHM (with blanks)

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#The forward recursion
# First, at t = 1
alpha(1,1) = y(1,Sext(1))  #This is the blank
alpha(1,2) = y(1,Sext(2))
alpha(1,3:N) = 0
for t = 2:T
    alpha(t,1) = alpha(t-1,1)*y(t,Sext(1))
    for i = 2:N
        alpha(t,i) = alpha(t-1,i-1) + alpha(t-1,i))
        if (skipconnect(i))
            alpha(t,i) += alpha(t-1,i-2)
        alpha(t,i) *= y(t,Sext(i))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

BACKWARD ALGORITHM WITH BLANKS

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#The backward recursion
# First, at t = T
beta(T,N) = 1
beta(T,N-1) = 1
beta(T,1:N-2) = 0
for t = T-1 downto 1
    beta(t,N) = beta(t+1,N)*y(t+1,Sext(N))
    for i = N-1 downto 1
        beta(t,i) = beta(t+1,i)*y(t+1,Sext(i)) + beta(t+1,i+1))*y(t+1,Sext(i+1))
        if (i<N-2 && skipconnect(i+2))
            beta(t,i) += beta(t+1,i+2)*y(t+1,Sext(i+2))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

COMPUTING POSTERIORS

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#Assuming the forward are completed first
alpha = forward(y, Sext)    # forward probabilities computed
beta  = backward(y, Sext)   # backward probabilities computed

#Now compute the posteriors
for t = 1:T
    sumgamma(t) = 0
    for i = 1:N
        gamma(t,i) = alpha(t,i) * beta(t,i)
        sumgamma(t) += gamma(t,i)
    end
    for i=1:N
        gamma(t,i) = gamma(t,i) / sumgamma(t)
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

COMPUTING DERIVATIVES

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#Assuming the forward are completed first
alpha = forward(y, Sext)    # forward probabilities computed
beta  = backward(y, Sext)   # backward probabilities computed

# Compute posteriors from alpha and beta
gamma = computeposteriors(alpha, beta)

#Compute derivatives
for t = 1:T
    dy(t,1:L) = 0 #Initialize all derivatives at time t to 0
    for i = 1:N
        dy(t,Sext(i)) -= gamma(t,i) / y(t,Sext(i))
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

CTC loss - mytorch/ctc_loss.py

```
def forward(self, logits, target, input_lengths, target_lengths):
    """CTC loss forward.

    Computes the CTC Loss.

def backward(self):
    """CTC loss backard.

    This must calculate the gradients wrt the parameters and return the
    derivative wrt the inputs, xt and ht, to the cell.
```

```
for b in range(B):
    # ------------------------------------------>
    # Computing CTC Loss for single batch
    # Process:
    #     Truncate the target to target length
    #     Truncate the logits to input length
    #     Extend target sequence with blank
    #     Compute forward probabilities
    #     Compute backward probabilities
    #     Compute posteriors using total probability function
    #     Compute expected divergence and store it in totalLoss
    # <------------------------------------------

    # ------------------------------------------>

    # Your Code goes here
    raise NotImplementedError
    # <------------------------------------------
```

```
for b in range(B):
    # ---------------------------------------->
    # Computing CTC Derivative for single batch
    # Process:
    #     Truncate the target to target length
    #     Truncate the logits to input length
    #     Extend target sequence with blank
    #     Compute derivative of divergence and store them in dY
    # <----------------------------------------

    # ---------------------------------------->

    # Your Code goes here
    raise NotImplementedError
    # <----------------------------------------
```

Beam Search - mytorch/search.py

```
def GreedySearch(SymbolSets, y_probs):
    """Greedy Search.

    Input
    -----
    SymbolSets: list
                all the symbols (the vocabulary without blank)

    y_probs: (# of symbols + 1, Seq_length, batch_size)
            Your batch size for part 1 will remain 1, but if you plan to use your
            implementation for part 2 you need to incorporate batch_size.

    Returns
    ------
    forward_path: str
                the corresponding compressed symbol sequence i.e. without blanks
                or repeated symbols.

    forward_prob: scalar (float)
                the forward probability of the greedy path

    """
```

```
def BeamSearch(SymbolSets, y_probs, BeamWidth):
    """Beam Search.

    Input
    -----
    SymbolSets: list
                all the symbols (the vocabulary without blank)

    y_probs: (# of symbols + 1, Seq_length, batch_size)
            Your batch size for part 1 will remain 1, but if you plan to use your
            implementation for part 2 you need to incorporate batch_size.

    BeamWidth: int
                Width of the beam.

    Return
    ------
    bestPath: str
            the symbol sequence with the best path score (forward probability)

    mergedPathScores: dictionary
                        all the final merged paths with their scores.

    """
```

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, BlankPathScore, PathScore =
                    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
```

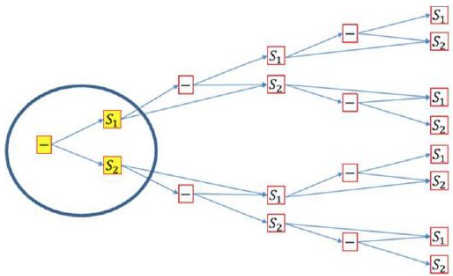
BEAM SEARCH InitializePaths: FIRST TIME INSTANT

```
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
```



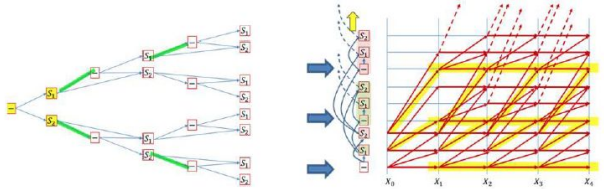
BEAM SEARCH: Extending with blanks

```
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 UpdatesPathsWithTerminalBlank
        # 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
```



BEAM SEARCH: Extending with symbols

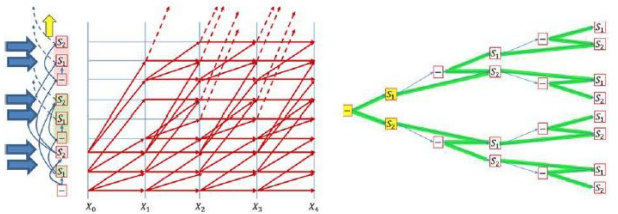
```
Global PathScore, BlankPathScore

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
```



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: Merging final paths

```
# Note : not using global variable here

function MergeIdenticalPaths(PathsWithTerminalBlank, BlankPathScore,
                             PathsWithTerminalSymbol, PathScore)

    # 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
```

Thank you!
Q & A