CTC(Connectionist Temporal Classification)

조희철

2021년 5월 10일

- Alex Graves et al. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks[1]
- CTC는 음성인식이나 문자인식(OCR)에서 target sequence와 output sequence의 길이가 다를 때(정확히는 output의 길이가 target보다 길 때), loss function을 정의하는 방법이다. target과 모델 output의 길이가 같지 않을 때, alignment가 맞지 않다고 한다.
- 'hello'라는 말하는 음성 파일로부터 'hello'라는 text를 만드는 것이 목적이다. 이때 모델은 한번에 'hello'를 만들어 내지 못하고 'hhhellloo'와 같은 형태로 문자열을 만들어 낸다. 같은 단어를 말할 때에도, 사람마다 음성 속도가 일정하지 않기 때문이다. 이 때, 연속된 'hhh'를 'h'로 합치는 것은 자연스럽지만, 연속된 character 'lll'이 문제이다. 'l'을 단순히 합쳐 버리면 'helo'가 되어 버린다. 그래서 합쳐야 하는 character와 합치지 않아야 하는 character를 구분할 필요가 있다.
- output이 만드는 연속된 같은 character는 하나로 합치고(merger), 실제 연속된 character는 다르게 처리하기 위해 'blank' 개념을 도입한다. 실제 연속된 character 사이에는 blank를 넣은 형태로 모델이 예측하게 만드는 것이다.
- 다시 말해, peuso character인 blank('-')를 도입하여 같은 character가 연속되는 경우를 다룰 수 있게 한다. 예를 들어, 'hello'의 'll , 'apple'의 'pp'는 각각 ('l', '-', 'l'), ('p', '-', 'p')와 같이 blank가 중간에 삽입된 형태로 모델이 예측하게 만든다. 물론 ('l', '-', '-' , 'l')과 같이 blank가 1개 이상 사이에 들어가는 것도 가능하다.
- blank 개념이 도입되면서, 주어진 GT가 될 수 있는 output(즉, prediction이 같은 output)이 여러개 될 수 있는데, 이 여러개의 output에 대한 확률을 모두 합하여 최대화 될 수 있도록 optimization을 수행한다.

Output	Probability	Prediction		Output	Probability	Prediction
aa	$0.4 \times 0.4 = 0.16$	a		ab	$0.4 \times 0 = 0$	ab
a-	$0.4 \times 0.6 = 0.24$	a		ba	$0 \times 0.4 = 0$	ba
bb	$0 \times 0 = 0$	b		b-	$0 \times 0.6 = 0$	b
-a	$0.6 \times 0.4 = 0.24$	a		-b	$0.6 \times 0 = 0$	b
- -	$0.6 \times 0.6 = 0.36$	-				

표 1: CTC loss: GT(target)가 'a'라면 Prediction이 'a'가 될 수 있는 모든 output의 확률의 합이 최대화 될 수 있도록 optimization하면 된다. 이 경우 prediction이 'a'가 되는 output은 모두 3가지가 있다(aa, a-,-a). 이렇게 prediction이 GT가 되는 모든 경우를 찾아 확률을 합해야 하는데, 이를 위해서 dynamic programming 기법이 사용된다.

[1] https://www.cs.toronto.edu/~graves/icml_2006.pdf, 참고자료: https://distill.pub/2017/ctc/

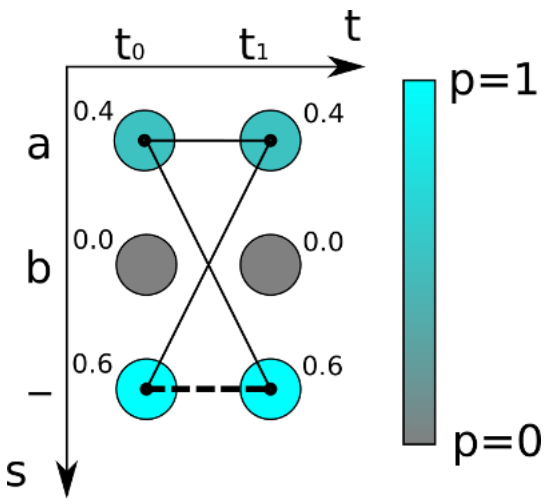


그림 1: CTC Loss: 전체 vocabulary가 {a,b} 2개만 있다고 가정하자. 여기에 black '-'를 더해 모델은 각 time step 별로 {a,b,'-'}에 대한 확률을 예측한다. vocabulary-size는 blank를 포함하여 3이고, time step(length)는 2이다. 이 time step은 입력되는 data와 모델의 구조에 따라 결정된다. 모델이 만들어 내는 3×2의 확률과 결합해서 loss를 계산해야 하는 target(=Groud Truth)의 길이는 time step보다 짧다. 그래서 alignment 문제가 발생한다.

- 주어진 GT를 만들 수 있는 모든 output의 확률 합을 구하기 위해 Dynamic Programming 기법이 사용된다. Dynamic Programming을 위해서 순환식을 만드는 것이 필요하다. 먼저 모든 valid path(GT가 되는 path)를 찾는 방법을 알아보자(좀 더 정확히는, 모든 valid path를 찾는 것이 아니고, path의 생성 확률 합을 구하는 것이다).

- 그림(2)과 그림(3)는 이동 규칙을 예를 들어서 설명하고 있다. 그림(2), 그림(3)에서는 모델의 output time step을 각각 5, 8로 가정하고 있다. 그림(2)을 먼저 살펴보자.

egg → -e-g-g- → 7×5 표를 만들어 path를 찾는다.
apple → -a-p-p-l-e- → 11×8 표를 만들어 path를 찾는다.

- GT(target) 'egg'에 blank를 넣어 새롭게 만든 sequence는 '-e-g-g-'가 되고, $c_s, (s = 1, 2, \cdots, 7)$은 각각 아래와 같이 된다. 'egg'가 길이 $n = 3$이므로, 시작/중간/끝에 blank를 넣으면 길이 $2n + 1 = 7$이 된다.

$$c_1 = \text{'-'}, c_2 = \text{'e'}, c_3 = \text{'-'}, c_4 = \text{'g'}, c_5 = \text{'-'}, c_6 = \text{'g'}, c_7 = \text{'-'}$$

- 7×5크기의 표가 만들어지고, 표에서 이동 규칙을 적용하여 path를 만들면 된다.

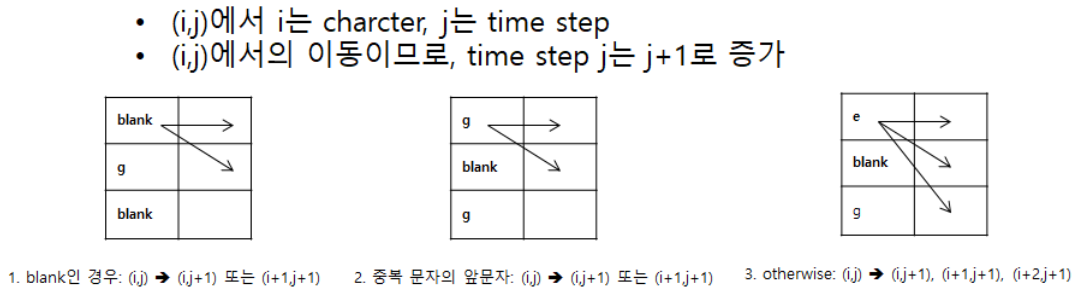


그림 2: time step이 5이고, target 단어 'egg'가 주어져 있다. 'egg'로 부터 7개의 문자열이 만들어지므로 7×5 크기의 표가 만들어 진다.
1. 이동 규칙 준수하면서 path를 만들면 된다. 이동 규칙을 준수한 path 모두가 유효한 것은 아니다. 즉, target과 일치하는 것은 아니라는 말이다.
2. 출발점이 $(1, 1)$ 또는 $(2, 1)$이고, 마지막 도착점이 $(6, 5)$ 또는 $(7, 5)$이면 만들어지는 단어가 'egg'가 되는 필요충분 조건이 된다. 다시 말해, 이동 규칙을 지키면서 $(6, 5)$ 또는 $(7, 5)$에 도달할 했다는 것은 blank 제거 후, 'egg'가 된다는 것을 의미한다.

	t_1	t_2	t_3	t_4	t_5
c_1('-')	→	→	→	↘	
c_2('e')				↘	
c_3('-')	↘				↘
c_4('g')		↘			
c_5('-')			↘		
c_6('g')				↘	→
c_7('-')					

표 2: 'egg'에 대한 path Table이 7×5인 이유: 7은 target 'egg'의 길이 3으로 부터 $2 \times 3 + 1 = 7$이 되기 때문이다. 5는 입력 data(waveform)의 길이와 모델 구조에 의해 결정되었다. smaple로 제시된 2개의 path는 각각 '- - - e -', 'e g - g g'에 해당한다. 2개 모두 이동 규칙이 준수되었지만, 1개만 target과 일치한다.

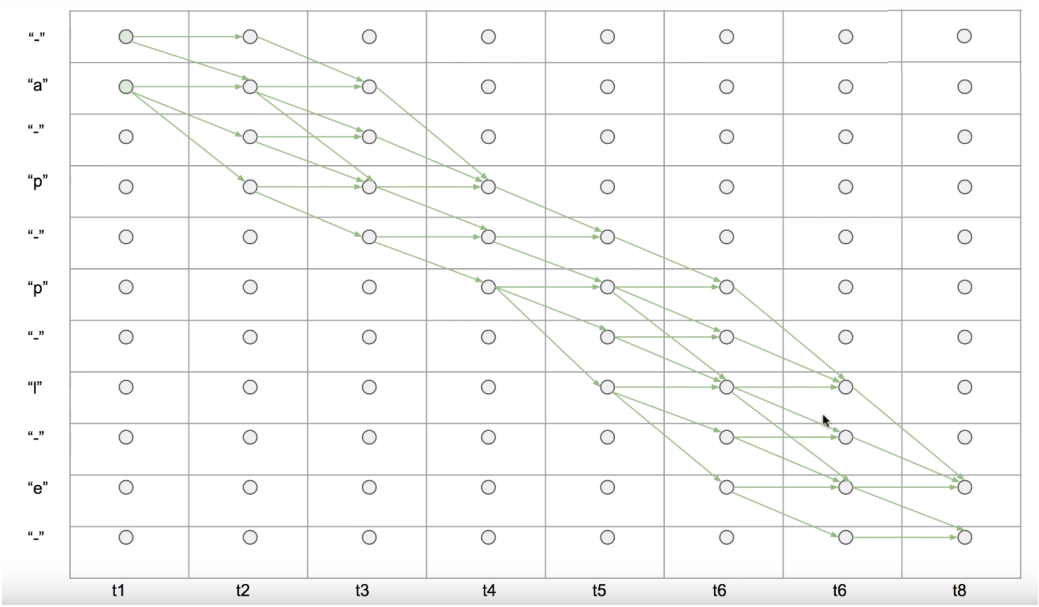


그림 3: CTC Loss: model output time step은 8이라 가정한다. Prediction이 target 'apple'이 될 수 있는 valid path를 보여준다. 주어진 target character 사이에 black를 넣어 새로운 sequence를 만든다. GT('apple')의 길이가 $n = 5$이므로, blank를 삽입한 sequence('-a-p-p-l-e-')의 길이는 $2n + 1 = 11$이 된다.

0. 모든 가능한 path를 찾기 위해, 이동 규칙을 만들면 문제가 쉬워진다. 이동 규칙을 만들어 보자.

1. 현재 위치가 (i, j)라고 할 때, 이동 규칙에 대하여 알아보자. blank인 경우: $(i, j + 1)$ 또는 $(i + 1, j + 1)$로 이동 가능

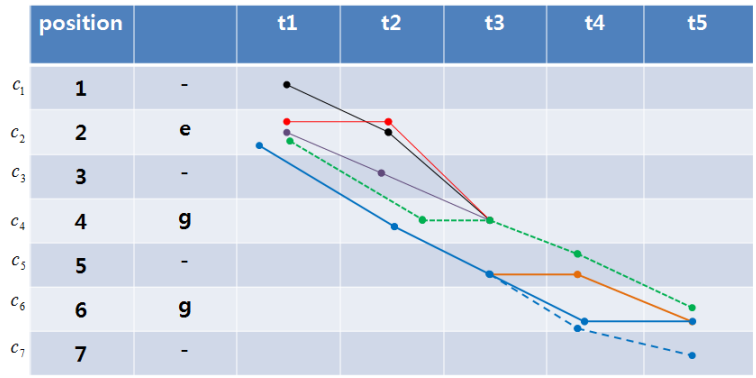
2. blank가 아니지만, 다음 character가 현재의 character와 같은 경우(중복된 character의 앞 character): 이 경우의 이동 규칙은 blank와 동일.

3. 그외: $(i, j + 1)$, $(i + 1, j + 1)$, $(i + 2, j + 1)$로 이동 가능.

4. 이런 규칙으로 이동했을 때, 오른쪽 최하단 위/아래 2칸에 도달한 path만이 character를 모두 포함하는, (즉, balnk 제거 후 Groun Truth가 되는) valid path가 된다.

5. 다시 말하지만, 우리의 최종 목적은 모든 valid path가 생성될 확률을 구해서 합치는 것이다. 이를 위하여 dynamic programming 기법을 활용하면, 모든 valid path 확률의 합을 구할 수 있다.

♠ Forward, Backward Computation



확률	t1	t2	t3	t4	t5	
a	0.351405	0.223719	0.178717	0.249302	0.221976	
e	0.180852	0.375489	0.486084	0.427556	0.544759	
g	0.068204	0.246327	0.108099	0.201085	0.114415	
blank	0.399539	0.154465	0.2271	0.122057	0.11885	

	t1	t2	t3	t4	t5	확률
"-eg-g"	0.399539	0.375489	0.108099	0.122057	0.114415	0.000226
"e-g-g"	0.180852	0.154465	0.108099	0.122057	0.114415	0.000042
"eeg-g"	0.180852	0.375489	0.108099	0.122057	0.114415	0.000103
"egg-g"	0.180852	0.246327	0.108099	0.122057	0.114415	0.000067
"eg-g-"	0.180852	0.246327	0.2271	0.201085	0.11885	0.000242
"eg-gg"	0.180852	0.246327	0.2271	0.201085	0.114415	0.000233
"eg--g"	0.180852	0.246327	0.2271	0.122057	0.114415	0.000141
						0.001054

그림 4: 'egg'에 대한 모든 valid path(time step = 5)가 모두 그려져 있다.

1. GT 'egg'의 character 사이에 blank를 끼워넣어서 새로운 sequence '-e-g-g-' 만든다.
2. 예를 들어 path '-e-g-'는 단어 'eg'가 되므로 'egg'와 다르다. path 'e-g-g'는 'egg'가 되므로 유효한 path이다.
3. valid path를 모두 찾아 보면 7개가 있다. '-eg-g','e-g-g','eeg-g','egg-g', 'eg-g-', 'eg-gg', 'eg–g'
4. vocabulary는 {a,e,g,-} 4개만 있다고 가정하자. 그러면, 모델은 4×5 크기의 확률 $\{p_t(c)\}$를 예측한다. 이 확률에 따라, path의 확률을 각각 구한 후, 모두 더하면 0.001054가 된다. CTC Loss $= -\log(0.001054) = 6.854927$
5. 일반적인 상황에서 모든 valid path를 찾기 위해서, dynamic programming 기법이 동원된다. 이를 위해서는 순환식이 필요하다.

- 그림 (4)에서는 모델이 예측한 확률을 직접 구한 path에 적용하여 path별 확률을 일일이 계산했다. 이제 일반화를 해보자.
- 모델의 output이 만들어내는, 각 time step t에 character c가 나올 확률을 $p_t(c)$라 하자. 즉 time t에서 character c가 나올 확률. 그림 (4)의 오른쪽 위에 있는 표가 바로 이 확률이다.
- 이제 순환식을 만들기 위해, $\alpha_t(s)$를 다음과 같이 정의하자.

$$\begin{aligned} \alpha_t(s) &= \text{the total probability of all subpaths whose prefixes end up with symbol } c_s \\ &\quad \text{at s-th position in the sequence at time } t \\ &= (1,1) \text{ 또는 } (2,1) \text{를 시작하고, } (t,s) \text{를 지나는 path의 확률 합.} \\ \alpha_1(s) &= 0 \text{ if } s \geq 3 \end{aligned}$$

- 그림 (4)에서 $\alpha_3(4)$는 $t = 3$에서 $c_4(=\text{'g'})$를 지나는 path들의 확률 합이 된다. 즉, 좌표 입장에서 $(3,4)$를 지나는 모든 path들의 확률 합. 좀 더 정확히하면, $c_4(=\text{'g'})$인 path는 모두 4개인데, $t = 1,2,3$까지 확률을 곱한 4개 값을 더한 것이 된다.

$$\begin{aligned} & 0.399593 \times 0.375489 \times 0.108099 \quad + \quad 0.180852 \times 0.154465 \times 0.108099 \\ & + 0.180852 \times 0.375489 \times 0.108099 \quad + \quad 0.180852 \times 0.246327 \times 0.108099 \\ = \; & 0.016217301 + 0.003019785 + 0.007340772 + 0.004815663 \\ = \; & 0.0313935207 \end{aligned}$$

- $\alpha_t(s)$는 t에서 character가 c_s인 모든 path 확률의 합이다. 우리가 구해야 하는 것은

$$\alpha_5(6) + \alpha_5(7)$$

- 이제 순환식(recursive relation)을 구해보자. 순환식을 이용하면, path를 직접 찾기 않고 확률을 계산할 수 있다. 이동 규칙을 만들 때, 출발점 기준으로 정의했는데, 순환식은 도착점 기준으로 이동 규칙을 적용하면 된다.

$$\alpha_t(s) = \begin{cases} \Big(\alpha_{t-1}(s-1) + \alpha_{t-1}(s)\Big) p_t(c_s), & \text{if } s = \text{ blank or 바로 앞에 같은 character} \\ \Big(\alpha_{t-1}(s-2) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s)\Big) p_t(c_s) & \text{otherwise} \end{cases}.$$

- 모든 t, s 대하여 $\alpha_t(s)$ 값을 구해보자. 순환식으로 부터 $t = 1, 2, \cdots$ 순으로 순차적으로 모든 s에 대하여 값을 구해나갈 수 있다. 전형적인 dynamic programming 방법이다.

- P =Prob('egg') $= \alpha_5(6) + \alpha_5(7)$, CTC loss $=$ L $:= -\log P = -\log\big(\alpha_5(6) + \alpha_5(7)\big) = 6.854927$

	t_1	t_2	t_3	t_4	t_5
1	$\alpha_1(1)$	$\alpha_2(1)$	$\alpha_3(1)$	$\alpha_4(1)$	$\alpha_5(1)$
2	$\alpha_1(2)$	$\alpha_2(2)$	$\alpha_3(2)$	$\alpha_4(2)$	$\alpha_5(2)$
3	0	$\alpha_2(3)$	$\alpha_3(3)$	$\alpha_4(3)$	$\alpha_5(3)$
4	0	$\alpha_2(4)$	$\alpha_3(4)$	$\alpha_4(4)$	$\alpha_5(4)$
5	0	$\alpha_2(5)$	$\alpha_3(5)$	$\alpha_4(5)$	$\alpha_5(5)$
6	0	$\alpha_2(6)$	$\alpha_3(6)$	$\alpha_4(6)$	$\boldsymbol{\alpha_5(6)}$
7	0	$\alpha_2(7)$	$\alpha_3(7)$	$\alpha_4(7)$	$\boldsymbol{\alpha_5(7)}$

	t_1	t_2	t_3	t_4	t_5
1	$\boldsymbol{\beta_1(1)}$	$\beta_2(1)$	$\beta_3(1)$	$\beta_4(1)$	0
2	$\boldsymbol{\beta_1(2)}$	$\beta_2(2)$	$\beta_3(2)$	$\beta_4(2)$	0
3	$\beta_1(3)$	$\beta_2(3)$	$\beta_3(3)$	$\beta_4(3)$	0
4	$\beta_1(4)$	$\beta_2(4)$	$\beta_3(4)$	$\beta_4(4)$	0
5	$\beta_1(5)$	$\beta_2(5)$	$\beta_3(5)$	$\beta_4(5)$	0
6	$\beta_1(6)$	$\beta_2(6)$	$\beta_3(6)$	$\beta_4(6)$	$\beta_5(6)$
7	$\beta_1(7)$	$\beta_2(7)$	$\beta_3(7)$	$\beta_4(7)$	$\beta_5(7)$

표 3: rowwise 방향으로 $\alpha_t(s)$를 모두 구하면 된다. 순환식을 수립하고, bottom up 방식으로 구하는 전형적인 dynamic programming이다. $P = \alpha_5(6) + \alpha_5(7) = \beta_1(1) + \beta_1(2)$

forward	t1	t2	t3	t4	t5	
blank	0.399539347	0.061715032	0.014015486	0.001710685	0.000203315	
1(e)	0.180851739	0.217930377	0.135931223	0.064110632	0.035856738	
blank		0.027935348	0.055836117	0.023406497	0.01040142	
2(g)		0.044548601	0.031393521	0.044874355	0.015147566	
blank		0	0.010116989	0.005066638	0.005935492	
2(g)		0	0	0.002034377	0.000812462	
blank		0	0	0	0.000241786	0.001054248

backward	t1	t2	t3	t4	t5	
blank	0.000226477	0	0			0.001054248
1(e)	0.000827772	0.000566844	0			
blank	0.001602245	0.000233184	0	0		
2(g)	0.000473187	0.003777046	0.001509616	0		
blank	0.002240429	0.003160752	0.01382387	0.013965117		
2(g)	0.000201588	0.002446778	0.006638647	0.04690616	0.114414957	
blank	0.000203315	0.000508874	0.003294417	0.014506456	0.118850102	

그림 5: 순환식으로 구한 $\alpha_t(s), \beta_t(s)$. forward, backward 결과가 같은 것을 확인할 수 있다.

- 지금까지 살펴본 $\alpha_t(s)$를 forward variable이라 한다. path에 대한 확률을 앞에서부터 구하지 않고, 뒤에서부터 계산하는 것도 가능하다. 그래서 대칭적으로 backward variable $\beta_t(s)$를 정의할 수도 있다.

$$\begin{aligned} \beta_t(s) &= \text{the total probability of all subpaths whose suffix start with symbol } c_s \\ &\quad \text{at s-th position in the sequence at time } t \\ \beta_5(s) &= 0 \text{ if } s \le 5 \end{aligned}$$

- $\beta_t(s)$의 순환식은 다음과 같다.

$$\beta_t(s) = \begin{cases} \Big(\beta_{t+1}(s+1) + \beta_{t+1}(s)\Big) p_t(c_s), & \text{if } s = \text{ blank or 바로 뒤에 같은 character} \\ \Big(\beta_{t+1}(s+2) + \beta_{t+1}(s+1) + \beta_{t+1}(s)\Big) p_t(c_s) & \text{otherwise} \end{cases}.$$

- $\alpha_t(s), \beta_t(s)$는 방향만 반대일 뿐이므로, 계산 방법은 동일하다.

- 정리 하면,

$$\text{Prob('egg')} = P = \alpha_5(6) + \alpha_5(7) = \beta_1(1) + \beta_1(2) = \sum_s \frac{\alpha_t(s)\beta_t(s)}{p_t(c_s)} \text{ for all } t$$

forward	t1	t2	t3	t4	t5	
blank	0.399539347	0.061715032	0.014015486	0.001710685	0.000203315	
e	0.180851739	0.217930377	0.135931223	0.064110632	0.035856738	
blank		0.027935348	0.055836117	0.023406497	0.01040142	
g		0.044548601	0.031393521	0.044874355	0.015147566	
blank		0	0.010116989	0.005066638	0.005935492	
g		0	0	0.002034377	0.000812462	
blank					0.000241786	0.001054248

backward	t1	t2	t3	t4	t5	
blank	0.000226477	0	0			0.001054248
e	0.000827772	0.000566844	0			
blank	0.001602245	0.000233184	0	0		
g	0.000473187	0.003777046	0.001509616	0		
blank	0.002240429	0.003160752	0.01382387	0.013965117		
g	0.000201588	0.002446778	0.006638647	0.04690616	0.114414957	
blank	0.000203315	0.000508874	0.003294417	0.014506456	0.118850102	

그림 6: CTC loss를 forward(왼쪽), backward(오른쪽) 방식으로 각각 계산한 결과가 동일(0.001054248)함을 확인할 수 있다.

alpha*beta/p	t1	t2	t3	t4	t5
blank	0.000226	-	-	-	-
e	0.000828	0.000329	-	-	-
blank	-	0.000042	-	-	-
g	-	0.000683	0.000438	-	-
blank	-	-	0.000616	0.000580	-
g	-	-	-	0.000475	0.000812
blank	-	-	-	-	0.000242
	0.001054248	0.001054248	0.001054248	0.001054248	0.001054248

← 모두 같은 값

그림 7: 각각의 time t에 대하여, $\sum_s \frac{\alpha_t(s)\beta_t(s)}{p_t(c_s)}$를 계산하면 동일한 확률이 계산됨을 알 수 있다. 식은 $\alpha_t(s), \beta_t(s)$를 곱하면 $p_t(c_s)$가 두번 곱해지기 때문에, 한번 나누어준 것이다. t에 상관없이 모든 열의 값($P_1 = P_2 = \cdots = P_5$)이 같기 때문에, 미분을 구할 때는 구하는 t에서의 P_t를 미분하면 된다.

♠ Gradient for Back-Propagation

- forward, backward variable을 결합하면 back-propagation에 필요한 gradient 계산이 가능하다. 모든 t, c에 대하여 Loss L을 $p_t(c)$로 미분해 보면,

$$\begin{aligned}\frac{\partial L}{\partial p_t(c)} &= -\frac{\partial \log P}{\partial p_t(c)} \\ &= -\frac{1}{P}\frac{\partial P}{\partial p_t(c)} \\ &= -\frac{1}{P}\frac{\partial P_t}{\partial p_t(c)}\end{aligned}$$

- 따라서 $\frac{\partial P_t}{\partial p_t(c)}$를 계산할 수 있으면 된다. 즉,

$$P_t = \sum_s \frac{\alpha_t(s)\beta_t(s)}{p_t(c_s)}$$

을 $p_t(c)$로 미분하면 된다. $c_s \neq c$인 s의 $\alpha_t(s), \beta_t(s)$는 $p_t(c)$와 무관하기 때문에, 다음 식을 미분하면 된다.

$$\sum_{c_s=c} \frac{\alpha_t(s)\beta_t(s)}{p_t(c)} \quad (\text{즉}, c_s = c\text{가 되는 } s\text{들만 더해준다.})$$

- summation 속의 식에 대한 미분은 1차식의 미분에 불과한데[2], 다시 $\alpha(s), \beta(s)$로 표현하면 다음과 같이 된다.

$$\sum_{c_s=c} \frac{\alpha_t(s)\beta_t(s)}{p_t(c)^2} = \sum_{c_s=c} \frac{\alpha_t(s)\beta_t(s)}{p_t(c)^2} = \frac{1}{p_t(c)^2}\sum_{c_s=c} \alpha_t(s)\beta_t(s)$$

- 정리하면,

$$\frac{\partial L}{\partial p_t(c)} = -\frac{1}{P} \times \frac{1}{p_t(c)^2} \times \sum_{c_s=c} \alpha_t(s)\beta_t(s)$$

- 지금까지 CTC loss를 계산하는 과정을 살펴보았는데, 실제 계산에서는 1보다 작은 확률들의 곱이 나오기 때문에 underflow가 발생할 수 있다. 이런 underflow를 방지하기 위한 기법들이 사용되어야 한다.

---

[2] $\frac{\alpha_t(s)\beta_t(s)}{p_t(c)} = \frac{\alpha_t(s)}{p_t(c)}p_t(c) \times \frac{\beta_t(s)}{p_t(c)}p_t(c) \times \frac{1}{p_t(c)} = \frac{\alpha_t(s)}{p_t(c)} \times \frac{\beta_t(s)}{p_t(c)} \times p_t(c)$.
$\frac{\alpha_t(s)}{p_t(c)}, \frac{\alpha_t(s)}{p_t(c)}$ are independent of $p_t(c)$.

alpha*beta	t1	t2	t3	t4	t5
blank	0.000090	-	-	-	-
e	0.000150	0.000124	-	-	-
blank	-	0.000007	-	-	-
g	-	0.000168	0.000047	-	-
blank	-	-	0.000140	0.000071	-
g	-	-	-	0.000095	0.000093
blank	-	-	-	-	0.000029

Loss=-log(P) 에 대한 softmax미분					
	t1	t2	t3	t4	t5
a	- 0.6950636	-	-	-	-
e	- 4.3415521	- 0.8310830	-	-	-
g	-	- 2.6303934	- 3.8469782	- 2.2385051	- 6.7356193
blank	-	- 0.2589682	- 2.5721933	- 4.5050331	- 1.9296947

그림 8: 왼쪽 값들은 $\alpha_t(s)\beta_t(s)$를 구한 것이다. 오른쪽은 각각의 character에 대하여, $-\frac{1}{P}\frac{1}{p_t(c)^2}\sum_{c_s=c}\alpha_t(s)\beta_t(s)$를 구한 결과이다. 예를 들어, 'g'가 두번 나오기 때문에 두곳의 값을 더한 후, p_2('g') 제곱으로 나누고, 여기에 $-\frac{1}{P}$를 곱했다. 오른쪽 값은 Loss에 대한 확률(softmax 값)의 gradient에 해당한다.

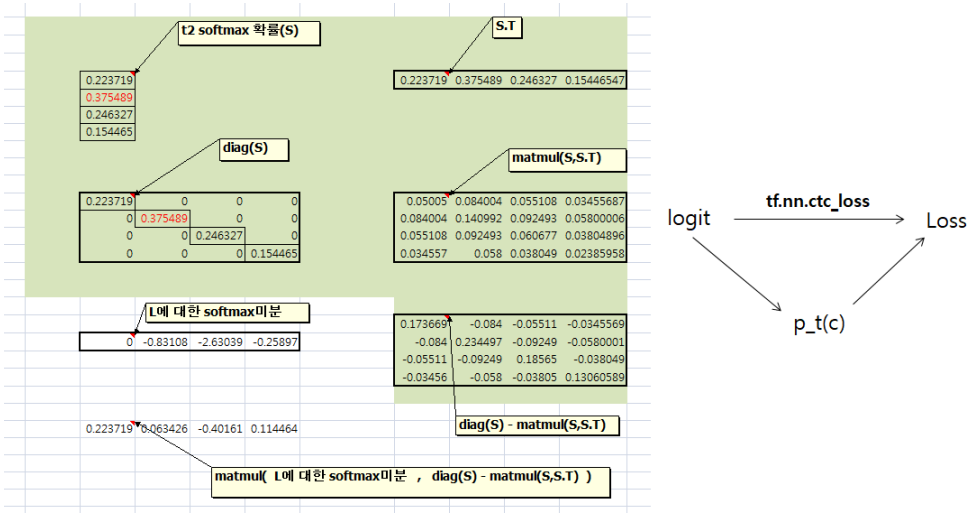


그림 9: Tensorflow의 tf.nn.ctc_loss는 logits를 입력으로 받아들인 후, softmax를 내부적으로 취하여 loss를 계산한다. 지금까지 구한 softmax 값에 대한 gradient 값과 비교하기 위해서는 softmax에 대한 Jacobian matrix를 구해야 한다. t를 고정하고, softmax에 대한 Jacobian matrix를 구해보자.

1. S를 softmax 확률로 두자. 예: 그림 (4)에서 $S = (0.223719, 0.375489, 0.246327, 0.154465)^T$
2. softmax에 대한 Jacobian matrix[3]는 $\mathrm{diag}(S) - SS^T =: J$
3. 그림 (8)에서 구한 softmax에 대한 gradient 행렬 (예: $(0, -0.8311, -2.6304, -0.2589)$)과 J를 곱해주면 logits에 대한 gradient가 된다.
4. 최종 결과 $(0.223719051, 0.063426442, -0.401609322, 0.11446383)$를 얻게되는데, Tensorflow에서 구현 결과와 일치함을 알 수 있다.

```
batch_size=2
output_T=5
target_T=3
num_class = 4

x = np.arange(40).reshape(batch_size,output_T,num_class).astype(np.float32)
x = np.random.randn(batch_size,output_T,num_class)
x = np.array([[[ 0.74273746,  0.07847633, -0.89669566,  0.87111101],
        [ 0.35377891,  0.87161664,  0.45004634, -0.01664156],
        [-0.4019564 ,  0.59862392, -0.90470981, -0.16236736],
        [ 0.28194173,  0.82136263,  0.06700599, -0.43223688],
        [ 0.1487472 ,  1.04652007, -0.51399114, -0.4759599 ]],

       [[-0.53616811, -2.025543  , -0.06641838, -1.88901458],
        [-0.75484499,  0.24393693, -0.08489008, -1.79244747],
        [ 0.36912486,  0.93965647,  0.42183299,  0.89334628],
        [-0.6257366 , -2.25099419, -0.59857886,  0.35591563],
        [ 0.72191422,  0.37786281,  1.70582983,  0.90937337]]]).astype(np.float32)

xx = tf.convert_to_tensor(x)
xx = tf.Variable(xx)
logits = tf.transpose(xx,[1,0,2])

yy = np.random.randint(0,num_class-1,size=(batch_size,target_T))  # low=0, high=3 ==> 0,1
yy = np.array([[1, 2, 2],[1, 0, 1]]).astype(np.int32)
```

```
zero = tf.constant(0, dtype=tf.int32)
where = tf.not_equal(yy, zero)
indices = tf.where(where)
values = tf.gather_nd(yy, indices)
targets = tf.SparseTensor(indices, values, yy.shape)

loss = tf.nn.ctc_loss(labels=targets,inputs=logits,sequence_length=[output_T]*batch_size)

optimizer = tf.train.GradientDescentOptimizer(learning_rate=1)
gradient = optimizer.compute_gradients(loss)
prob = tf.nn.softmax(xx,axis=-1)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
l = sess.run(loss)
g = sess.run(gradient[0][0])
p = sess.run(prob)
```

```
In [2]: g
Out[2]:
array([[[ 0.35140473, -0.6043253 ,  0.06820415,  0.18471655],
        [ 0.22371903,  0.06342658, -0.40160912,  0.11446385],
        [ 0.17871664,  0.48608422, -0.3077556 , -0.35704485],
        [ 0.24930191,  0.4275561 , -0.24904504, -0.4278127 ],
        [ 0.22197618,  0.54475874, -0.6562402 , -0.11049433]],

       [[ 0.32429516, -0.42046958,  0.51874053, -0.4225659 ],
        [ 0.1660185 , -0.4430562 ,  0.32442492, -0.04738733],
        [ 0.18140681, -0.17293978,  0.19122487, -0.19969228],
        [ 0.20436302, -0.04891258,  0.20998912, -0.3654397 ],
        [ 0.1788914 , -0.7739314 ,  0.47851837,  0.11652097]]],
      dtype=float32)
```

그림 10: Tensorflow에서 구한 logits에 의한 Loss 미분값. Tensorflow ctc loss함수는 character의 마지막 index를 blank로 받아들인다.

CTC loss는 입력 data 전체가 주어져 있는 Encoder에서 사용한다. data 전체가 주어져 있기 때문에, bidirectional rnn 구조도 가능하다. Tensorflow에서 output sequence를 만들기 위해서는 `tf.nn.ctc_greedy_decoder`, `tf.nn.ctc_beam_search_decoder`를 사용하면 된다.

[3] https://mattpetersen.github.io/softmax-with-cross-entropy

♠ Avoiding Underflows

- 1보다 작은 값인 확률을 반복적으로 곱하다보면, underflow가 발생할 수 있다. 논문에서는 underflow를 방지하는 방법을 제시하고 있다.

$$C_t := \sum_s \alpha_t(s), \quad \hat{\alpha}_t(s) := \frac{\alpha_t(s)}{C_t}$$

- t에 대해서 순차적으로 적용하여 $\{\hat{\alpha}_1(s)\}, \{\hat{\alpha}_2(s)\}, \{\hat{\alpha}_3(s)\}, \cdots$ 를 계산한다. 좀 더 상세한 과정을 살펴보면 다음과 같다.
  - $\{\alpha_1(s)\}$를 계산한 후 C_1이 구해지고, C_1으로 rescaling하여 $\{\hat{\alpha}_1(s)\}$를 구한다.
  - $\{\hat{\alpha}_1(s)\}$로부터 $\{\alpha_2(s)\}$가 계산된 후, C_2가 구해지고, C_2로 rescaling하여 $\{\hat{\alpha}_2(s)\}$를 구한다.
  - 이런 과정을 반복하여 모든 C_t를 계산할 수 있다.
- 이렇게 C_t가 계산되면, 최종 loss값은 다음과 같다.

$$-\log P = \sum_t \log C_t$$

- backward에 대하여 D_t를 정의하고, 같은 방법으로 계산이 가능하다.

$$D_t := \sum_s \beta_t(s), \quad \hat{\beta}_t(s) := \frac{\beta_t(s)}{D_t}$$

forward	t1	t2	t3	t4	t5	
blank	0.688396767	0.175262387	0.056675552	0.012115057		
1(e)	0.311603233	0.618892949	0.549676045	0.454031068		
blank	0	0.079332631	0.225789007	0.165764652		
2(g)	0	0.126512033	0.126948511	0.317799883		
blank	0	0	0.040910885	0.035881893		
2(g)	0	0	0	0.014407446	0.770655592	
blank	0	0	0	0	0.229344408	
Ct	0.580391086	0.606710487	0.702279802	0.570994696	0.007466179	
-log Ct	0.544053117	0.499703559	0.353423376	0.560375359	4.897371926	6.854927337

backward	t1	t2	t3	t4	t5	
blank	0.214822748	0	0	0		
1(e)	0.785177252	0.053008391	0	0		
blank		0.02180615	0	0		
2(g)		0.353210258	0.059747621	0		
blank		0.295577573	0.547121381	0.185268463		
2(g)		0.228810338	0.262744506	0.622281386	0.490493336	
blank		0.04758729	0.130386492	0.192450151	0.509506664	
Dt	0.098587967	0.423226671	0.33519912	0.323141981	0.233265059	
-log Dt	2.316806065	0.859847377	1.093030535	1.129663483	1.455579878	6.854927337

그림 11: CTC rescaling: rescaling을 이용한 Loss 계산.

- 논문에서는 $\hat{\alpha}_t(s), \hat{\beta}_t(s)$을 이용한 미분으로, softmax를 취하기 이전의 logits(unnomralized outputs) u_k^t에 대한 Loss의 미분식이 제시되어 있다.

$$\begin{aligned} y_c^t &= p_t(c) = \frac{\exp u_c^t}{\sum_{c'} \exp u_{c'}^t} \\ Z_t &:= \sum_c \sum_{c_s=c} \frac{\hat{\alpha}_t(s)\hat{\beta}_t(s)}{y_{c_s}^t} = \sum_c \frac{1}{y_c^t} \sum_{c_s=c} \hat{\alpha}_t(s)\hat{\beta}_t(s) \\ -\frac{\partial \log P}{\partial u_c^t} &= y_c^t - \frac{1}{y_c^t Z_t} \sum_{c_s=c} \hat{\alpha}_t(s)\hat{\beta}_t(s) \end{aligned}$$

logit에 대한 미분	t1	t2	t3	t4	t5
a	0.3514048	0.2237191	0.1787167	0.2493019	0.2219762
e	-0.6043255	0.0634264	0.4860842	0.4275561	0.5447587
g	0.0682042	-0.4016093	-0.3077557	-0.2490451	-0.6562406
blank	0.1847166	0.1144638	-0.3570452	-0.4278129	-0.1104943

그림 12: $\hat{\alpha}_t(s), \hat{\beta}_t(s)$를 이용하여 logits에 대한 Loss 미분 결과. 그림 (10)에 있는 결과와 일치한다.