Your Neural Network for NLP is Probably Wrong

Why You Need to Mask More Than You Think

Brian Lester

Interactions

February, 27, 2020

Slides

This QR code has a link to the slides if you want to follow along.

Me

- Work on NLP at Interactions
- My specialization is in Deep Learning
- Maintain Mead-Baseline

- This is my "Why you should listen to me" slide
- I do NLP at interactions where we use NLP to "facilitate customer care interactions" aka build better chat bots
- My specialization is DL and We use DL for most models so I have a lot of experience
- I help maintain Mead-Baseline, our open-source Deep Learning modeling software so I have a lot of experience writing Neural Networks for NLP
- I have batched some complex operations for Mead-Baseline
  - Batched the CRF
  - Batched Beam Search
- I have tracked down a lot of hidden batch instability problems
- I am trying to get more involved with research so if anyone needs help with some idea, especially if the implementation is tricky, hit me up.

Binary Logistic Regression

$n :=$ number of features

$c :=$ number of class $:= 1$

$f \in \mathbb{R}^n$

$w \in \mathbb{R}^n$

$$s = \sum_{i=0}^{n} f_i * w_i$$

- First I'm going to build up batching through building examples by starting with Binary LR
- In binary LR we have a vector representing features and a vector of the same size representing weights
- (Normally in NLP we have sparse features where we don't have a actual vector of features, we would use something like a hashmap instead, but inside a neural network everything is dense vectors so we will use dense vectors here because it is just a pedagogical example)
- We take the sum of the features weighted by the weight to create a score which we call a logit.
- This operation is called the dot product.
- Normally we then us a function like sigmoid to create a final probability (and then calculate losses and gradients) but we don't need to worry about that for this example.

Binary Logistic Regression

```
s = np.dot(f, w)
```

- In this example we are representing our feature and weights as numpy vectors
- W can use the dot product function which to calculate the sum of the element wise products of the vectors.

Binary Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$
$$w = \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix}$$
$$s =$$

- Here we can see our feature vector and our weight vector
- Now lets look at the mechanics of what is happening in our dot product calculation

Binary Logistic Regression

$$
\begin{aligned}
f &= \begin{bmatrix} \boxed{1} & 2 & 3 & 4 \end{bmatrix} \\
w &= \begin{bmatrix} \boxed{5} & 6 & 7 & 8 \end{bmatrix} \\
s &= 5
\end{aligned}
$$

- We multiply these terms together and add the result into the s

Binary Logistic Regression

$$
\begin{aligned}
f &= \begin{bmatrix} 1 & \boxed{2} & 3 & 4 \end{bmatrix} \\
w &= \begin{bmatrix} 5 & \boxed{6} & 7 & 8 \end{bmatrix} \\
s &= 17
\end{aligned}
$$

- And it continues on like this.

Binary Logistic Regression

$$f = [\ \boxed{1\ \ 2\ \ 3\ \ 4}\ ]$$
$$w = [\ \boxed{5\ \ 6\ \ 7\ \ 8}\ ]$$
$$s = 70$$

- Here we highlight the elements in these arrays that interact with each other in our dot product.

Multi-class Logistic Regression

$$n := \text{number of features}$$
$$c := \text{number of class}$$
$$f \in \mathbb{R}^n$$
$$W \in \mathbb{R}^{n \times c}$$
$$s \in \mathbb{R}^c$$
$$\forall_{j \in c}\, s_j = \sum_{i=0}^{n} f_i * W_{ij}$$

- We now extend to multi-class LR
  - In binary LR we are deciding between to things for example Positive or Negative sentiment
  - In multi-class LR we are deciding between multi classes, for example Science, Technology, Politics, cooking, and video games for document classification
- We still have a feature vector for some document.
- We now have multiple weight vectors, one for each class
- We pack our weight vectors into a single matrix of size n (the number of features we have) by c (which is the number of classes we have).
- Now our logits are a vector, we have a score for each class
- for each class we do this dot product between the feature vector and one of the weight vectors
- Once we have the score we normally turn it into probabilities with the softmax function but we can skip this for this example.

Multi-class Logistic Regression

```
s = []
for w in W.T:
    s.append(np.dot(f, w))
s = np.array(s)
```

- Here we see code that calculates the logits how I described it before.
- where we explicitly do the dot between the feature and each weight vector in turn
- We turn the results into a vector at the end

Multi-class Logistic Regression

```
s = np.dot(f, W)
```

- But it turns out we can actually do this in a single dot product! Lets see why

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} & \end{bmatrix}$$

- Here we can see our feature vector again and see our new weight matrix
- Note how the weight matrix is two weight vectors next to each other
- The first column is the same weight vector we saw earlier
- Lets explore how the elements of the feature vector and weight matrix are combined

Multi-class Logistic Regression

$$f = \begin{bmatrix} \boxed{1} & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} \boxed{5} & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} \boxed{5} & \end{bmatrix}$$

- Here we can see that we multiply an element from the feature vector and the first column of the weight matrix
- We save this value into the score vector

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 17 & \end{bmatrix}$$

- And it goes on like this...

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & \end{bmatrix}$$

- We see here that the value in this first spot of the logits was calculated between the feature vector and the first column in the weight matrix

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 9 \end{bmatrix}$$

- Here we see the elements of the feature vector and the second column weight matrix are combined and saved into the second element of the logit vector

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 29 \end{bmatrix}$$

- And it goes on like this...

Multi-class Logistic Regression

$$f = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \end{bmatrix}$$

- We see here that the value in the second spot of the logits was calculated between the feature vector and the second column in the weight matrix
- So we can see here that we can calculate the logits for multiple classes at once between there is no interaction between the columns of the matrix in vector-matrix multiplication.

Batched Multi-class Logistic Regression

$$n := \text{number of features}$$
$$c := \text{number of class}$$
$$b := \text{number of examples}$$
$$F \in \mathbb{R}^{b \times n}$$
$$W \in \mathbb{R}^{n \times c}$$
$$S \in \mathbb{R}^{b \times c}$$
$$\forall_{k \in b} \forall_{j \in c} S_{kj} = \sum_{i=0}^{n} F_{ki} * W_{ij}$$

- We just saw how you can calculate the scores for multiple classes for a single example in a single vector-matrix multiply. Now we will see how you can actually do this for multiple examples at once.
- Here we see Batched Multi-class Logistic Regression.
- This means that we are doing multi-class LR but will process multiple examples at once
- Like before we still have a weight matrix of n by c
- In this case we have b examples
- Now we have stacked multiple feature vectors into a feature matrix with b rows where each row represents a different feature vector.
- Now our scores are a matrix were each row is the scores for an example and each column is the score for some class
- We could calculate this by doing the dot product of each feature vector and the weight matrix and stacking the results into our score matrix

Batched Multi-class Logistic Regression

```
S = []
for f in F:
    S.append(np.dot(f, W))
S = np.stack(S)
```

- Here we see a straight forward way to do this, where we loop over the features and calculate the scores for each example separately.
- The dot product of the feature vector and weight matrix returns a vector
- We collect these vectors and stack them into a single array

Batched Multi-class Logistic Regression

```
S = np.dot(F, W)
```

- But we can actually do one better, we can calculate the whole thing, all classes for all examples in a single shot!
- Lets take a closer look

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} & \end{bmatrix}$$

- Here we see our new feature matrix is two feature vectors stacked on top of each other
- Our weight matrix is still a n by c matrix of weight vectors standing next to each other
- We are now doing matrix matrix multiplication

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 5 & \\ & \end{bmatrix}$$

- Here we see the values for the first class of the first example are computed using the values in the first row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 17 & \\ & \end{bmatrix}$$

- Here we see the values for the first class of the first example are computed using the values in the first row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & \\ & \end{bmatrix}$$

- Here we see the values for the first class of the first example are computed using the values in the first row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 9 \end{bmatrix}$$

- Here we see the values for the second class of the first example are computed using the values in the first row of the feature vector and the second column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 29 \\ & \end{bmatrix}$$

- Here we see the values for the second class of the first example are computed using the values in the first row of the feature vector and the second column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ & \end{bmatrix}$$

- Here we see the values for the second class of the first example are computed using the values in the first row of the feature vector and the second column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 65 & \end{bmatrix}$$

- Here we see the values for the first class of the second example are computed using the values in the second row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 149 & \end{bmatrix}$$

- Here we see the values for the first class of the second example are computed using the values in the second row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 382 & \end{bmatrix}$$

- Here we see the values for the first class of the second example are computed using the values in the second row of the feature vector and the first column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 382 & 117 \end{bmatrix}$$

- Here we see the values for the second class of the second example are computed using the values in the second row of the feature vector and the second column of the weight matrix

Batched Multi-class Logistic Regression

$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 382 & 117 \end{bmatrix}$$

- Here we see the values for the second class of the second example are computed using the values in the second row of the feature vector and the second column of the weight matrix

Batched Multi-class Logistic Regression

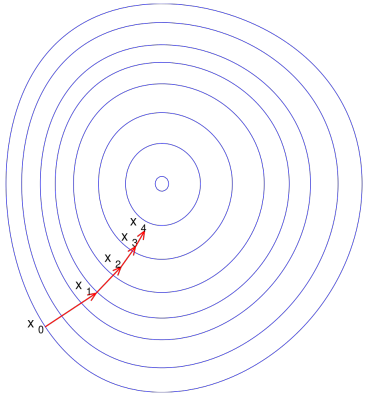
$$F = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$W = \begin{bmatrix} 5 & 9 \\ 6 & 10 \\ 7 & 11 \\ 8 & 12 \end{bmatrix}$$

$$s = \begin{bmatrix} 70 & 110 \\ 382 & 614 \end{bmatrix}$$

- And finally the last elements uses the two unused rows and columns.
- So here we have seen that we can stack independent examples together just like we could stack together independent weight vectors to calculate the scores for multiple examples at once.
- This works because the rows and columns in matrix-matrix multiplication all act independently of each other.
- This means we can put otally unrelated data an be next to each other and they won't interact as it flows through your network
- Running multiple examples like this is what we mean by batching.

Full Gradient Descent



Pros:

- This is the true gradient
- You step is guaranteed to yield better performance on all examples

Cons:

- It is slow
- You need to run each example before you update any parameters

- The way we train these models is with something called gradient descent
- This basically means run an example through, calculate how wrong the predicted answer is (the loss), and use the gradient (the derivative) of the parameters with respect to this loss to calculate how to update the parameters.
- In some cases people do full gradient descent where they calculate the answers for each example and get the gradient for everything
- this is good because this is the true gradient, this is the best step you can take to do better next time
- This is slow because you have to weight to run the network on all your data before you update any parameters
- In this picture we can see the big strides in the right direction
- Image from here

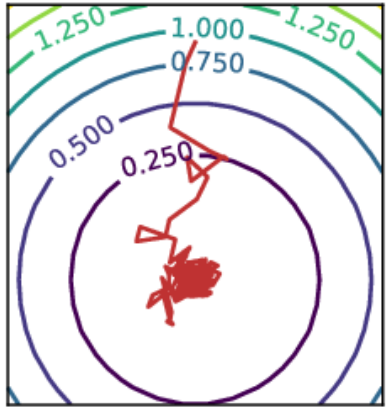
Stochastic Gradient Descent

Pros:

- You get to update your parameters each step.
- You can get better generalization because to the randomness.

Cons:

- Your gradient could be wrong.
- The gradient for one example can cause a change that is detrimental to another example.



- On the opposite end of the spectrum we have stochastic gradient descent
- We calculate a gradient and update parameters after each example
- Talk about pros and cons
- In this image we can see how much more random the gradients are
- Image from here

Mini-Batched Gradient Descent

Pros:

- You get to update your parameters more often.
- You can a better approximation of your gradient.

Cons:

- Your need to run the examples in a batch which can be tricky.
- Minibatch size is now a hyperparameter.

- This is a happy middle ground between the two
- We calculate the gradient for a small batch of examples and update
- Talk about pros and cons
- We can run this mini-batch together via batching

Gradient Accumulation?

- We saw before that we want to have batches for training dynamics
- Why do you have to actually run things at once as a single batch?
- Can't you just run each example individually and accumulate the gradients in memory and apply them when you get enough?
- Given that I am giving a whole talk on why batching is hard shouldn't you try to avoid it?

- Gradient Accumulation is used when your networks are really large and can't fit on a single GPU
- But we have a good reason to actually batch and that is...

Speed

- Image from here
- We showed before that we can express batched operations as matrix matrix multiplications
- Lucky for us humanity has gotten really good at doing these matmuls fast
- We have specialized libraries and even specialized devices like GPUs are designed to do these in a parallel and fast way
- We can stand on the shoulders of gaming to train models really fast

Batching in NLP

[The dog ran very fast]

- Now that we know batching is a good idea and you are all really jazzed up about it, I'm going to rain on your parade a bit.
- The problem is that batching is pretty hard in NLP
- We saw before that batching involves stacking multiple vectors of the same shape into a matrix
- In NLP we often have examples that have different lengths
  - Because sentences have different lengths
  - Because words have different number of characters in them
- We can't just batch any given pair of sentences.

Batching in NLP

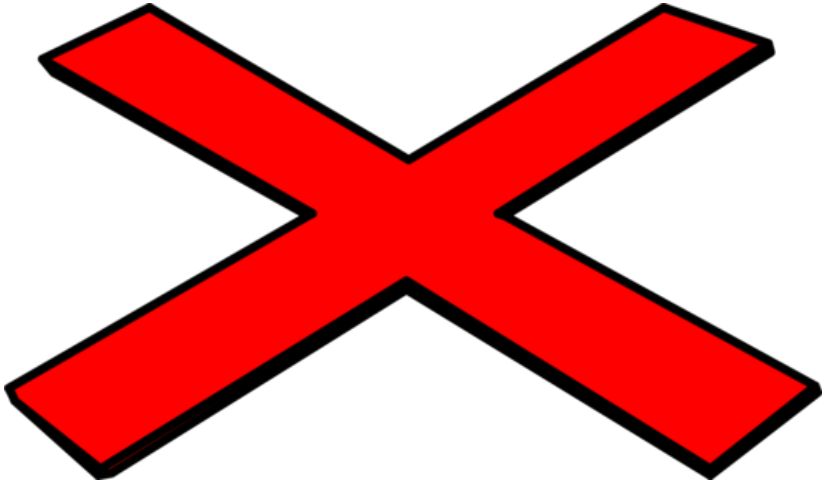
$$\begin{bmatrix}\text{The} & \text{dog} & \text{ran} & \text{very} & \text{fast}\end{bmatrix}$$

$$\begin{bmatrix}\text{The} & \text{cat} & \text{slept}\end{bmatrix}$$

- Now that we know batching is a good idea and you are all really jazzed up about it, I'm going to rain on your parade a bit.
- The problem is that batching is pretty hard in NLP
- We saw before that batching involves stacking multiple vectors of the same shape into a matrix
- In NLP we often have examples that have different lengths
  - Because sentences have different lengths
  - Because words have different number of characters in them
- We can't just batch any given pair of sentences.

Batching in NLP

$$\begin{bmatrix} \text{The} & \text{dog} & \text{ran} & \text{very} & \text{fast} \\ \text{The} & \text{cat} & \text{slept} & & \end{bmatrix}$$



- Here we can't batch these examples because they are different lengths, this would create a ragged array, we need it to be regular.
- Image from here

Introducing <PAD>

$$\begin{bmatrix} \text{The} & \text{dog} & \text{ran} & \text{very} & \text{fast} \\ \text{The} & \text{cat} & \text{slept} & <\text{PAD}> & <\text{PAD}> \end{bmatrix}$$

- Now lets look at how we can handle batching things of different lengths.
- We are going to use a new symbol call <PAD>.
- We add this to the end of shorter sentence so that all the sentences in a batch have the sample length.
- Now that they are all the same length we can turn them into a matrix.

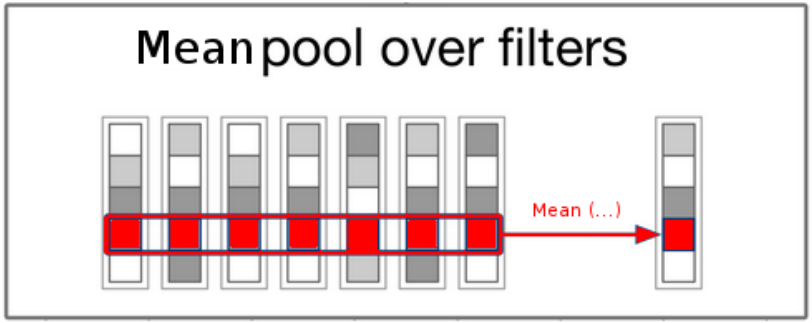
The lengths vector

$$\begin{bmatrix} \text{The} & \text{dog} & \text{ran} & \text{very} & \text{fast} \\ \text{The} & \text{cat} & \text{slept} & <\text{PAD}> & <\text{PAD}> \end{bmatrix}$$

$$L = \begin{bmatrix} 5 & 3 \end{bmatrix}$$

- Now that some of our examples are longer because if this <PAD >we need to track how long the actual data is
- We store this in a lengths vector
- This will be a vector of the same length as the number of items in the batch
- The value will be the number of non padded elements in that example
- You can actually have padding on every example in the batch.
- This means the length of the physical batch is longer than the maximum value in the lengths vector
- This is normally an artifact of being lazy with your data preprocessing, a bit sloppy, and can result in extra computation being done.
- Because doing this is wasteful we won't really consider this situation.

Mean Pooling



- As a warm up we are going to look at Mean Pooling.
- The main use-case of mean pooling is in the Neural Bag of Words model.
- We embed each word resulting in a matrix that is sentence length by embedding size
- We do a mean over each time, so each feature is mean-ed separately
- This results in a sentence vector that we can then use for classification

Mean Pooling

$$\begin{bmatrix} 1 & 10 & 8 & 17 & 13 & 17 \end{bmatrix} = \frac{66}{6} = 11.0$$

- Mean pooling gives us a clear example of how padding can get us in trouble
- Side Note: In most of these examples there is actually another dimension of features that these operations are actually acting on but we are going to ignore it and only really look at a single feature in this dimension for simplicity
- Here we see the mean pooling for a vector of length 6
- Here we see the mean pooling for a length of 4 (shorter)

Mean Pooling

$$\begin{bmatrix} 1 & 10 & 8 & 17 & 13 & 17 \end{bmatrix} = \frac{66}{6} = 11.0$$

$$\begin{bmatrix} 22 & 24 & 9 & 13 \end{bmatrix} = \frac{68}{4} = 17.0$$

- Mean pooling gives us a clear example of how padding can get us in trouble
- Side Note: In most of these examples there is actually another dimension of features that these operations are actually acting on but we are going to ignore it and only really look at a single feature in this dimension for simplicity
- Here we see the mean pooling for a vector of length 6
- Here we see the mean pooling for a length of 4 (shorter)
- How we see that in order to batch we need to pad the shorter vector
- Now our denominator is much larger so our result of mean-ing is smaller!
- Before I said that batching allows us to combine unrelated examples and calculate them in one short and get the same results as if you had calculated them independently
- When your padding goes wrong the results for an example will be dependent on the other things in the batch. When these padding cases information to bleed across examples this independence is broken and our calculations will be wrong.

Mean Pooling

$$\begin{bmatrix} 1 & 10 & 8 & 17 & 13 & 17 \end{bmatrix} = \frac{66}{6} = 11.0$$

$$\begin{bmatrix} 22 & 24 & 9 & 13 \end{bmatrix} = \frac{68}{4} = 17.0$$

$$\begin{bmatrix} 5 & 4 & 8 & 9 & 10 & 34 \\ 6 & 3 & 1 & 4 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \frac{66}{6} \\ \frac{68}{6} \end{bmatrix} = \begin{bmatrix} 11.0 \\ 11.\bar{3} \end{bmatrix}$$

- Mean pooling gives us a clear example of how padding can get us in trouble
- Side Note: In most of these examples there is actually another dimension of features that these operations are actually acting on but we are going to ignore it and only really look at a single feature in this dimension for simplicity
- Here we see the mean pooling for a vector of length 6
- Here we see the mean pooling for a length of 4 (shorter)
- How we see that in order to batch we need to pad the shorter vector
- Now our denominator is much larger so our result of mean-ing is smaller!
- Before I said that batching allows us to combine unrelated examples and calculate them in one short and get the same results as if you had calculated them independently
- When your padding goes wrong the results for an example will be dependent on the other things in the batch. When these padding cases information to bleed across examples this independence is broken and our calculations will be wrong.

Mean Pooling

```
>>> x
array([[ 1, 10,  8, 17, 13, 17],
       [22, 24,  9, 13,  0,  0]])
>>> np.mean(x, axis=1)
array([11.        , 11.33333333])
>>> lengths
array([6, 4])
>>> np.sum(x, axis=1) / lengths
array([11., 17.])
```

- We can solve this by explicitly using the lengths vector.
- We can manually do our sum and then divide by the actual length instead of the length of the padded vector
- This assumes that the padded value is zero. Soon we will see how to insure that.
- This sort of thing is important because if you don't do it then values will dependent on what else is in the batch.
- This problem is especially important in training vs production environments
  - when training it an example is almost always in a batch
  - When you are in a batch is is relatively rare to be the longest so being correct in the presence of padding is important
  - In prod you are often a single example and you are by definition the longest thing in the batch
  - This means that you need to have the same results with and without padding

Cross Entropy

$$L_{\text{cross-entropy}}(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$

- Out next example of where padding can get up to no good is in our loss functions
- Our normal loss is the cross entropy loss between two vectors
- y is the vector that has a our labels and $\hat{y}$ is out predicted labels
- cross entropy is a measure of the difference between two distributions
- The cross entropy is the sum of the true value times the log of the predicted values
- In classifiction where we output a probability distribution over classes we can use this loss function to train the model

Sparse Cross Entropy

$$L_{\text{cross-entropy}}(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i)$$

$$L_{\text{sparse-cross-entropy}}(\hat{y}, y) = -\log(\hat{y}_i[y])$$

- Because our labels are always a one hot vector (it only has a 1 at the location that represents the label index) most of these elements in this sum will be zero
- In fact the only value will be the probability for the predicted class
- We can short cut this calculation by just indexing the predicted vector with the true class index
- This is what we call sparse cross entropy

Token-Level (Sparse) Cross Entropy Loss

$T :=$ length of example

$V :=$ number of possible labels

$s \in \mathbb{R}^{T \times V}$

$l \in \mathbb{N}^{T}$

$$J = -\sum_{i}^{T} s_i[l_i]$$

- This idea of sparse cross entropy can be extend to the idea of a token level loss.
- This means that a decision is made for every token and we calculate the loss for the probability vector that is output for each token in a sentence.
- We will see soon how this loss at the padding positions can mess us up.
- Here we see the cross entropy loss where we have a matrix of scores. Each row represents the scores of a token and each column is the score for some class.
- Our loss is then the sum (or mean) of the score for each token.
- This kind of loss is often used for language modeling where the label is the index of the next word or else sequence tagging where the label is the tokens tag.

(Sparse) Cross Entropy Loss

$$
s = \begin{bmatrix} -0.34 & 0.1 & -0.11 & \cdots & 0.001 \\ 0.93 & -8.88 & -0.39 & \cdots & 0.12 \\ & & \vdots & & \\ -0.45 & 0.23 & 1.1 & \cdots & -0.3 \end{bmatrix}
$$

$$
l = \begin{bmatrix} 2 & 1 & 10 & 5 & 1 \end{bmatrix}
$$

$$
J = \begin{bmatrix} -0.11 & -8.88 & 0.28 & 0.43 & 0.23 \end{bmatrix}
$$

- Here we see some examples of score, labels, and the selected scores.

(Sparse) Cross Entropy Loss

$$s = \begin{bmatrix} -0.34 & 0.1 & -0.11 & \cdots & 0.001 \\ 0.93 & -8.88 & -0.39 & \cdots & 0.12 \\ & & \vdots & & \\ -0.45 & 0.23 & 1.1 & \cdots & -0.3 \end{bmatrix}$$

$$l = \begin{bmatrix} 2 & 1 & 10 & 5 & 1 \\ 5 & 6 & 3 & & \end{bmatrix}$$

$$J = \begin{bmatrix} -0.11 & -8.88 & 0.28 & 0.43 & 0.23 \\ -3.4 & 0.6 & 0.37 & -956.0 & 10000.0 \end{bmatrix}$$

- Here we see what can happen when you are batching and have a shorter sentence.
- When you are selecting the scores for the padding tokens they really could be anything
- We don't really want the network to have to learn to output a padding symbol after the normal length, that isn't a situation you would really have a use for.
- We also don't want random lucky values to make our loss look really good.
- We need a way to zero our the effects of these padded locations

Making a Mask

```
>>> lengths = np.array([5, 3, 4])
>>> steps = np.arange(np.max(lengths))
>>> steps
array([0, 1, 2, 3, 4])
>>> mask = (np.reshape(lengths, (-1, 1)) > steps).astype(np.int
>>> mask
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0]], dtype=int32)
```

- We will zero out these values by using a mask
- This mask has ones where the valid tokens are and a zero where the padding tokens are.
- Here we see how to create a mask.
- We create this with broadcasting.
- We create a vector of incrementing steps along the dimension that includes padding and compare them to the lengths vector.
- When the length is longer than the current steps we are still valid so the value is one.
- Once the step is longer than the length we are in the pad and will output zeros.

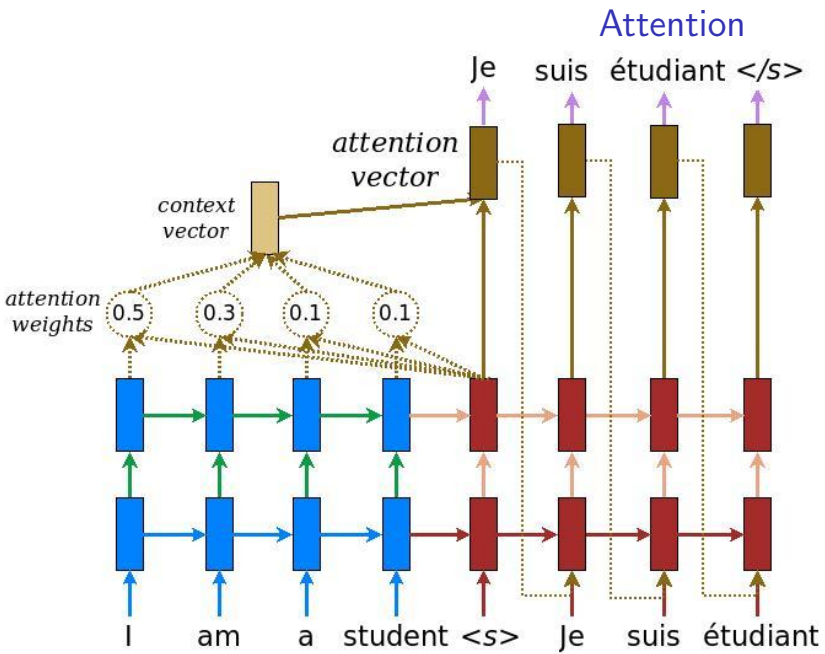
(Sparse) Cross Entropy Loss

$$J = \begin{bmatrix} -0.11 & -8.88 & 0.28 & 0.43 & 0.23 \\ -3.4 & 0.6 & 0.37 & -956.0 & 10000.0 \end{bmatrix}$$

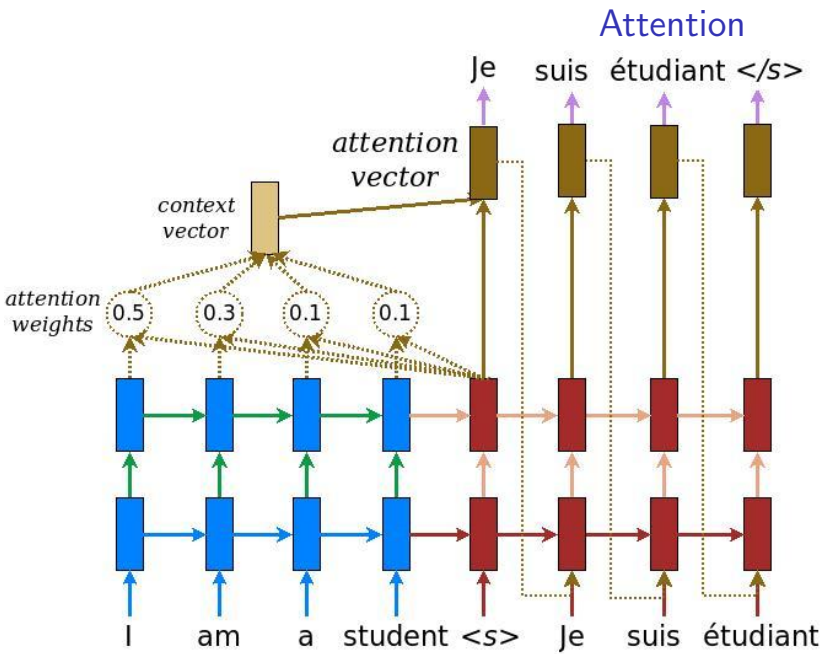
$$\text{mask} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$J * \text{mask} = \begin{bmatrix} -0.11 & -8.88 & 0.28 & 0.43 & 0.23 \\ -3.4 & 0.6 & 0.37 & 0 & 0 \end{bmatrix}$$

- Here we can use our mask to force the padding locations to zero.
- We can then sum the values (or mean them with the lengths if you want).
- Some toolkits like Pytorch can handle this for you, you give a value to the `padding_idx` parameter and it will ignore values where the label is that padding index
- Toolkits like tensorflow will probably make you do this our self.



- Attention is used to contextualize a vector relative to others.
- We have some vector here in the decoder and want to create a new vector that summarizes the information in these vectors created by the encoder that is relevant to decoder vector.
- We do this by creating a similarity score between the vector we care about and each vector in the memory bank of vectors
- We then normalize these values and create new vector by taking a weight average of the memory bank
- This is often used in seq2seq models to short cut information from the encoder into the decoder
- Image From Here



- In a more general format we have 3 sets of vectors
  - Query Vectors: the ones we are processing right now
  - Key Vectors: The ones we are doing the similarity score with
  - Value Vectors: The ones that are actually mixed together
- In some cases these vectors can be the same (for example RNN Seq2seq normally use $K = Q$) in other case they are either different or the same thing projected into different spaces
- The point of this is that the representation of a vector that is useful down stream might not be one that is similar to the query vectors so the keys can be the similar ones.
- Image From Here

Attention

$$q = \begin{bmatrix} — & q & — \end{bmatrix}$$

$$K = \begin{bmatrix} | & | & & | \\ k_1 & k_2 & \cdots & k_n \\ | & | & & | \end{bmatrix}$$

$$s = \text{sim}(q, K) = \begin{bmatrix} s_1 & s_2 & \cdots & s_n \end{bmatrix}$$

- First we calculate a score between the vectors
- There are multiple ways to calculate these similarities, the original paper used a small neural network, later architectures (notably the transformer) use the dot product to calculate the score.
- The transformer actually uses the scaled dot product where it divides the score by $\sqrt{d_k}$ because the dot product will get larger as the vectors get larger
- In this case we are just using a single vector for q but we can actually use a matrix Q to calculate the attention between multiple vectors at once.
- This vector version is common for RNN seq2seq models (where you have single vector at each timestep)
- The Matrix version is used in the transformer where we want to do attention for the whole sentence at once

Attention

$$s = \text{sim}(q, K) = \begin{bmatrix} 0.21 & -0.34 & \cdots & 0.55 \end{bmatrix}$$

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$\alpha = \text{softmax}(s) = \begin{bmatrix} 0.1684 & 0.0977 & \cdots & 0.2379 \end{bmatrix}$$

Numerically stable softmax blog post.

- Then we use the softmax so the score are all positive and they sum to one.
- The softmax is the done by exponentiating each element and then normalizing the elements.
- I have a blog post about how you can do that in a numerically stable way if you decide to roll your own implementation.
- We can then uses these scores to do a weight average.

Attention

$$V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_n \\ | & | & & | \end{bmatrix}$$

$$s = \text{sim}(q, K) = \begin{bmatrix} s_1 & s_2 & \cdots & s_n \end{bmatrix}$$

$$\alpha = \text{softmax}(s)$$

$$c = V * \alpha^T$$

- Then we do a weighted average of the value vectors based on these calculated (and normalized) scores.
- Now c is a combination of the vectors in V weighted by their attention scores which were calculated based on the similarity of the query vectors to q

Attention

$$K = \begin{bmatrix} | & | & & | & | \\ k_1 & k_2 & \cdots & \text{<PAD>} & \text{<PAD>} \\ | & | & & | & | \end{bmatrix}$$

$$s = \begin{bmatrix} s_1 & s_2 & \cdots & 0 & 0 \end{bmatrix}$$

- But what happens when we calculate the similarity between vectors that have padding on them?
- With most similarity metrics (dot/scaled dot product) we do get zeros for the similarity between a padding vector and a valid vector (assuming the pad is zeros)
- Is this enough so that our calculations will be correct?
- Would I have brought this up if it was?

Attention

```
>>> x = np.array([
        [0.32, 0.11, 0.83, 0.9, 0.4],
        [0.5, 0.2, -1.0, 0, 0]
    ])
>>> x
array([[ 0.32,  0.11,  0.83,  0.9 ,  0.4 ],
       [ 0.5 ,  0.2 , -1.  ,  0.  ,  0.  ]])
>>> softmax(x)
array([[0.15759942, 0.12774761, 0.26244893, 0.28147862, 0.17072542],
       [0.31476139, 0.23318098, 0.07023276, 0.19091244, 0.19091244]])
```

- Here we can see that when the scores going into the softmax are zero we will get non-zero scores out!
- This means that when we do the mixing of the value vectors were are going to include the padding vectors in our weighted average.
- What we need to have there be zeros **after** the softmax which will zero out the contribution of the padding.

Attention

```
>>> mask = make_mask(np.array([5, 3]))
>>> mask
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 0, 0]], dtype=uint8)
>>> attn = softmax(x) * mask
>>> attn
array([[0.15759942, 0.12774761, 0.26244893, 0.28147862, 0.17072542],
       [0.31476139, 0.23318098, 0.07023276, 0.        , 0.        ]])
>>> np.sum(attn, axis=1)
array([1.        , 0.61817513])
>>> softmax(x[1:np.newaxis, 0:3])
>>> array([[0.50917835, 0.3772086 , 0.11361305]])
```

- Ok, our first cut is just masking out those padding weights, now their values are zero so they won't contribute to the weighted average anymore.
- Is this ok?
- We can see that the values for this example don't sum to one any more, this solution isn't enough
- We we can also see that the values calculated for the sorter batch are very different then when that example is calculated by itself.
- Remember the whole point of batching is that we should be able to run multiple examples at once and get the same values as if we ran them through one at a time

Attention

```
>>> inv_mask = (1 - mask) * -1e9
>>> masked_x = (x * mask) + inv_mask
>>> masked_x
array([[ 3.2e-01,  1.1e-01,  8.3e-01,  9.0e-01,  4.0e-01],
       [ 5.0e-01,  2.0e-01, -1.0e+00, -1.0e+09, -1.0e+09]])
>>> softmax(masked_x)
array([[0.15759942, 0.12774761, 0.26244893, 0.28147862, 0.17072542],
       [0.50917835, 0.3772086 , 0.11361305, 0.        , 0.        ]])
>>> softmax(x[1:np.newaxis, 0:3])
>>> array([[0.50917835, 0.3772086 , 0.11361305]])
>>> softmax(masked_x).sum(axis=1)
array([1., 1.])
```

- We need to hack softmax to force it to output zeros for the padded values while the other values sum to one.
- We know that the softmax will exponentiate each element and we know that exponentiation by a negative number of will approach zero as the exponent grows.
- With a sufficiently negative number we can force an element to zero
- On this first line `(1 - mask)` will flip a mask of ones and zeros.
  - $1 - 1 = 0$ and $1 - 0 = 1$
  - This will mean that the ones are now where the pads and zero where the valid elements are
- Then we multiply in a very negative number so out mask is all zeros and this neg
- We multiply by the mask to make sure the padding locations are zero (not strictly necessary) then we add the inverse mask in.
- the inverse mask has all zeros for the valid elements so they don't change but it has $-1e^9$ for the pad so zero $+$ that negative is that negative
- Now when we do the softmax we see that out values match and they sum to one in each axis

Attention

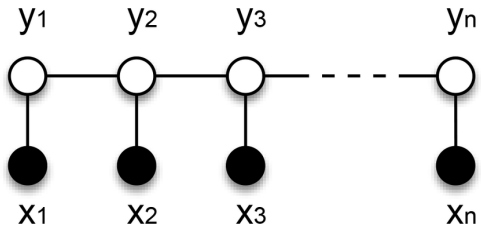
Encoder-Decoder Attention

Encoder Self-Attention

MaskedDecoder Self-Attention

- We have been looking at encoder-decoder attention where we compare a vector in the decoder to the values in the encoder.
- There is also self attention were we compare to all the tokens around us.
- There is also a special case of self attention where we can use a mask to enforce causality
- This mask would stop the attention from looking forward in time at words that haven't been generated yet.
- This mask works the same way as the length mask.
- Some times you need to combine these masks.
- Attention is an example where just masking out the padding value isn't enough, we actually have to manipulate the padding values in a specific way.
- Image is from here

Conditional Random Field



$$P(\hat{y}|x;\theta) = \frac{\exp(\sum_i \sum_j w_j f_j(\hat{y}_{i-1}, \hat{y}_i, x, i))}{\sum_{y' \in Y} \exp(\sum_i \sum_j w_j f_j(y'_{i-1}, y'_i, xi))}$$

Great Tutorial on CRFs

- This is our most complex example
- Image from Here

Conditional Random Field

$$t := \text{number of tokens}$$

$$c := \text{number of classes}$$

$$\alpha \in \mathbb{R}^S$$

$$e \in \mathbb{R}^{t \times c}$$

$$t \in \mathbb{R}^{c \times c}$$

$$\forall_{i \in t}\, \alpha_i = \text{logsumexp}(\alpha_{i-1} + e_i^T + t)$$

Conditional Random Field

```
>>> lengths = np.array([3, 1, 5])
>>> lengths
array([3, 1, 5])
>>> for i in range(np.max(lengths)):
...     new_alphas = calculate_step(alpha, emission, transistion)
...     mask = (i < lengths).astype(np.uint8)
...     print(mask)
...     alphas = (alpha * mask) + (new_alphas * (1 - mask))
[1 1 1]
[1 0 1]
[1 0 1]
[0 0 1]
[0 0 1]
```

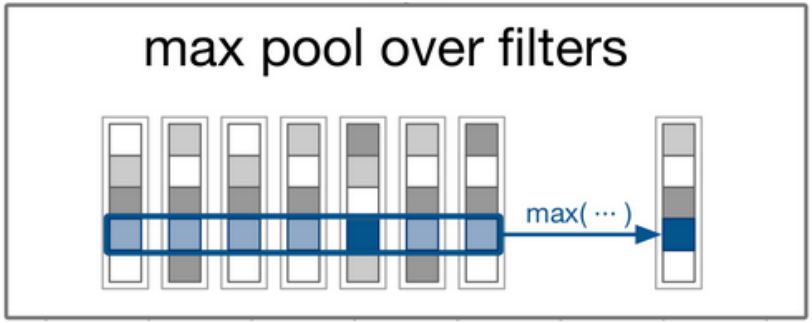
Conditional Random Field

```
>>> a
array([1, 2, 3, 4, 5])
>>> b
array([-1, -2, -3, -4, -5])
>>> mask = np.where(a >= 4, 1, 0)
>>> mask
array([0, 0, 0, 1, 1])
>>> (a * mask) + (b * (1 - mask))
array([-1, -2, -3,  4,  5])
```

CRF

This needs a lot more
CRF are used in sequence tagging to model dependencies in the output
At each step we calculate the possible scores or transitioning to any state from any state
Once we hit the end we don't want to overwrite our scores with transistors into the padding
The CRF gives us a new way to handle batches, we essentially have an if, only update alpha if we are still inside the valid lengths. We do an if by calculating the results of both branches (do nothing to the alpha and calculate the next step) and use a mask to combine them.
The Viterbi decoding works the same way

Max Pooling



- Max pooling is often applied across a feature dimension in the same way that mean pooling is
- Here we see a set of vectors, each one is a featured representation of a word in a sentence
- We do max pooling across a single feature dimension.
- We end up with a single vector where each feature is the largest value that feature took in the course of the sentence
- We can also do this at a word level where each vector represents a character in a word
- A common use case is to encode words with a Convolutional Network (which we will get to soon) and then aggregate the results with max pooling.
- Image from here

Max Pooling

$$x = \begin{bmatrix} 1 & 0 & 4 & 3 & 2 & 0 & 0 \end{bmatrix}$$
$$\text{lengths} = \begin{bmatrix} 5 \end{bmatrix}$$
$$\max(x) = 4$$

- Here we are zoomed in a single feature over time, these are the numbers in the blue highlighted area on the last slide
- Lets say these vectors we are zoomed in one just went through a ReLU $\max(0, x)$ so it will never have a negative number in it.
- This example has a length of 5 which means the last two zero features are padding
- Pop Quiz: Is it safe to do a max without thinking about padding?
- **YES**

Max Pooling

$$x = \begin{bmatrix} -3 & -1 & -4 & -3 & -2 & 0 & 0 \end{bmatrix}$$
$$\text{lengths} = \begin{bmatrix} 5 \end{bmatrix}$$
$$\max(x) = 0$$
$$\max(x[:5]) = -1$$

- What about a vector that looks like this?
- Lets say here we are looking at a feature of a word right after embedding.
- Your max pooling it going to output zero for this feature!
- Your network will never see a negative value for most examples while training! (most examples are padded)
- Your network will see these negative values at test time and have no idea what to do!
- How do we fix this?

Max Pooling

$$x' = \begin{bmatrix} -3 & -1 & -4 & -3 & -2 & -10000 & -10000 \end{bmatrix}$$
$$\text{lengths} = \begin{bmatrix} 5 \end{bmatrix}$$
$$\max(x') = -1$$

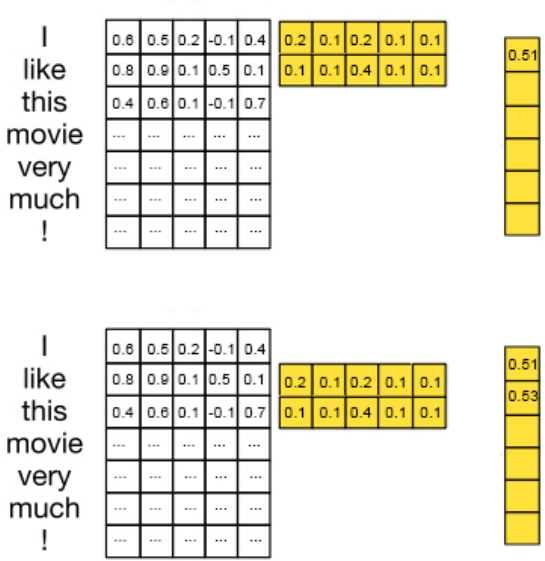
- We use the same trick we used in attention, by setting the pads so a very negative value we can ensure they will never be picked

Min Pooling

$$
\begin{aligned}
x &= \begin{bmatrix} 3 & 1 & 4 & 3 & 2 & 0 & 0 \end{bmatrix} \\
x' &= \begin{bmatrix} 3 & 1 & 4 & 3 & 2 & 10000 & 10000 \end{bmatrix} \\
\text{lengths} &= \begin{bmatrix} 5 \end{bmatrix} \\
\min(x) &= 0 \\
\min(x[:5]) &= 1 \\
\min(x') &= 1
\end{aligned}
$$

- This same thing applies in Min pooling, we where the padding can be selected as the minimum.
- We can solve this similarly but set the padding to a very high value.

Convolution 1D



- Convolutions are basically used to process soft n-grams.
- This is commonly used in text classification
- The conv will look at multiple words at a time, and create a new representation of the word contextualized by the words around it.
- Once we have contextualized the representation we will aggregate into a single vector with some sort of pooling.
- It is also common in taggers to use a convolution to create a representation of a word based on the characters.
- Image from here

Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 32 & 38 & 44 \end{bmatrix}$$

- So this shows us how a convolution works, we look at a window, scale the values by a learned weight matrix and sum them to create a new value for this position.
- Unfortunately here we can see that a convolution will actually create a new, shorter sentence
- We need to introduce a new form of padding that I will call conv-padding to distinguish from the padding we have been talking about before

Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & \boxed{5 & 6 & 7} & 8 \end{bmatrix}$$

$$\begin{bmatrix} 32 & \boxed{38} & 44 \end{bmatrix}$$

- So this shows us how a convolution works, we look at a window, scale the values by a learned weight matrix and sum them to create a new value for this position.
- Unfortunately here we can see that a convolution will actually create a new, shorter sentence
- We need to introduce a new form of padding that I will call conv-padding to distinguish from the padding we have been talking about before

Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 32 & 38 & 44 \end{bmatrix}$$

- So this shows us how a convolution works, we look at a window, scale the values by a learned weight matrix and sum them to create a new value for this position.
- Unfortunately here we can see that a convolution will actually create a new, shorter sentence
- We need to introduce a new form of padding that I will call conv-padding to distinguish from the padding we have been talking about before

Padded Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} \boxed{0 \;\; 4 \;\; 5} & 6 & 7 & 8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \boxed{23} & 32 & 38 & 44 & 23 \end{bmatrix}$$

- Here we can see how conv-padding can be used to stop the sequence from getting shorter
- Here we can see we are adding a padding of zeros on each side
- There are filtersz // 2 units on each side
- Another advantage of this conv-padding is that without padding there is on;y a single calculation that involves the first word. conv-padding makes it so that there is more information from the tokens on each end.
- We see here how we now get a vector that is the same length as the input (before padding)

Padded Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4 & 5 & 6 & 7 & 8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 23 & 32 & 38 & 44 & 23 \end{bmatrix}$$

- Here we can see how conv-padding can be used to stop the sequence from getting shorter
- Here we can see we are adding a padding of zeros on each side
- There are filtersz // 2 units on each side
- Another advantage of this conv-padding is that without padding there is on;y a single calculation that involves the first word. conv-padding makes it so that there is more information from the tokens on each end.
- We see here how we now get a vector that is the same length as the input (before padding)

Padded Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4 & 5 & 6 & 7 & 8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 23 & 32 & 38 & 44 & 23 \end{bmatrix}$$

- Here we can see how conv-padding can be used to stop the sequence from getting shorter
- Here we can see we are adding a padding of zeros on each side
- There are filtersz // 2 units on each side
- Another advantage of this conv-padding is that without padding there is on;y a single calculation that involves the first word. conv-padding makes it so that there is more information from the tokens on each end.
- We see here how we now get a vector that is the same length as the input (before padding)

Padded Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4 & 5 & 6 & 7 & 8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 23 & 32 & 38 & 44 & 23 \end{bmatrix}$$

- Here we can see how conv-padding can be used to stop the sequence from getting shorter
- Here we can see we are adding a padding of zeros on each side
- There are filtersz // 2 units on each side
- Another advantage of this conv-padding is that without padding there is on;y a single calculation that involves the first word. conv-padding makes it so that there is more information from the tokens on each end.
- We see here how we now get a vector that is the same length as the input (before padding)

Padded Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4 & 5 & 6 & 7 & 8 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 23 & 32 & 38 & 44 & 23 \end{bmatrix}$$

- Here we can see how conv-padding can be used to stop the sequence from getting shorter
- Here we can see we are adding a padding of zeros on each side
- There are filtersz // 2 units on each side
- Another advantage of this conv-padding is that without padding there is on;y a single calculation that involves the first word. conv-padding makes it so that there is more information from the tokens on each end.
- We see here how we now get a vector that is the same length as the input (before padding)

Batched Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 6 & 14 & 7 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 6 & 14 & 7 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 54 & 55 & 28 & 7 & 0 \end{bmatrix}$$

- In this example we are zoomed in on an example that has already been padded, the two zeros at the end
- Now were are going to pad it again because the conv-padding is applied to the whole batch at once
- As we slide the conv we can see that at the end we have spots where the only part of the window is on the valid tokens but it is writing into the padding section of the output vector.
- This is introducing a new value what could be selected by the max pool during training time that would never happen at test time.

Batched Convolution 1D

$$w = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

When we were already padded our convolution will bleed into the padding

$$\begin{bmatrix} 0 & 6 & 14 & 7 & 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 54 & 55 & 28 & 7 & 0 \end{bmatrix}$$

When we do this convolution by itself

$$\begin{bmatrix} 0 & 6 & 14 & 7 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 54 & 55 & 28 \end{bmatrix}$$

- In this example we are zoommed in on an example that has already been padded, the two zeros at the end
- Now were are going to pad it again because the conv-padding is applied to the whole batch at once
- As we slide the conv we can see that at the end we have spots where the only part of the window is on the valid tokens but it is writing into the output vector.
- This is introducing a new value what could be selected by the max pool during training time that would never happen at test time.

Batched Convolution 1D

These Convolutions are commonly used for:

- Encoding word n-grams which are then max pooled to create sentence representations for text classification
- Encoding character n-grams and max pooling to create word representations. These “char compositional” features are common for sequence tagging
- Creating a windowed context for tagging

- Does this matter? Yes, when I first noticed this but I trained a tagger on ‘Conll 2003‘ and the first thing I trained have slightly different results based on if it was run on a single example of batches so it seems very prevalent.
- Elmo has this batch instability bug but given their cavalier attitude to the RNN hidden states I think the engineering in Elmo is pretty from the hip so they probably don’t care
- For the last use case I don’t think we really need to care, padding in a later stage (CRF or the token level loss) should save us.
- ”But how do I fix this one Brian?” This is one is pretty easy you just need to zero out the padded values after they go through the conv. If you are following it with maxpool you might want to just jumpr right to masking the pad values with a very negative number

Unit Tests

```
def test_batch_stability():
    ex1 = generate_example(length=random.randint(5, 10))
    ex2 = generate_example(length=random.randint(10, 15))
    res1 = network(ex1, lengths=[len(res1)])
    res2 = network(ex2, lengths=[len(res2)])
    batch, lengths = batch_examples(ex1, ex2)
    res = network(batch, lenghts=lengths)
    batch1, batch2 = extract_results(res, lengths)
    np.testing.assert_allclose(batch1, ex1)
    np.texting.assert_allclose(batch2, ex2)
```

An example of these tests in our CRF

- The easiest way to avoid these errors is a unittest, There is a nice recipe for these batch instability tests
  1. Generate two examples with different lengths
  2. Run each example through the thing you want to test and record results
  3. Batch the two examples together
  4. Run the batch through the thing you want to test
  5. Extract the values that represent each example and compare them to the values from running solo

Conclusion

- Almost nothing is a safe when you are padding.
- You should be passing lengths around for most things.
- Some frameworks help manage padding and you should use them when you can.
- These bugs are hard to find so you should be writing unittests designed to check for them.

- As mentioned pytorch losses can handle this for you with the `padding_idx` parameter.
- Most RNN implementations take a length parameter and handle this for you.
- DyNet has an autobatching feature where they basically have a computation graph compiler.
  - You can write forward passes that are not batched and the compiler will batch them on the fly.
  - Unfortunately DyNet isn't very active these days.

Contact

- Slides
- Twitter
- Github
- Mead-Baseline

- Here is a link to these slides which are in a git repo with the latex to make them.
- This is my twitter, feel free to talk to me about batching or any NLP stuff.
- This is my github which has a lot of (i think) cool NLP stuff on it
- And this is a link to Mead-Baseline, we currently have a lot of layers that don't have these batch stability tests. So if this talked interested you and you want to try your hand at it make a PR maybe?
- Thank you, any questions?