Introduction to machine learning algorithms (V0.98)

David Levinthal
Performance lead
Microsoft Cloud Services Infrastructure

The rapid rise in the successful use of machine learning (ML) algorithms raises the issue of what an optimal architecture for ML acceleration should look like. To help such an investigation, some discussion of the numerical operations of the dominant ML networks is required as a starting point. What follows will hopefully shed some light on this.

The two dominant ML network architectures, at the time of this writing, are convolutional neural networks (CNNS) and recurrent neural networks (RNNs). CNNs are currently dominantly used for 2-dimensional image recognition like surveillance, self-driving cars and drones and tagging which selfies have a cat. RNNs are dominantly used for sequential input (text, voice) recognition and manipulation respectively. As convolution is the basis for FFTs it is not surprising that CNNs are also useful in decomposing speech waveform patterns into phonemes, from which they can then be processed by RNNs into language and processed. These two techniques represent the dominant approaches in DNN applications.

Learning

Machine learning algorithms apply simple linear operations (among others) to the data as it flows through the paths defined by the network

Output_j = ∑ weight_j_i X input_i + bias_j (sum over I, the repeated index)

The weights and biases are adjusted during learning/training, typically by stochastic gradient descent. This is done with a tagged set of training data where the correct answers for each input set are known. The error between the networks evaluation of the training tag information and the training data's values for the tags is minimized. This is done through an evaluation of the derivatives of the error with respect to the weights and biases used everywhere throughout the network. This process is referred to as training. More on this later (see backpropagation section).

Once the weights have been trained the network can be used with some reliability on untagged data. This is referred to as inference.

In both inference and training the data sets can be processed in batches. During training the size of the batch (aka mini-batch) is chosen based on training performance and stability. During inference the batch size may be limited by the input request rate and the required response latency. Frequently a batch size of 1 may be all that is allowed.

The batch (mini-batch) sizes in practice tend to be small even for training, tens to perhaps a few hundred when networks of platforms are used together. They are usually referred to as mini-batches. A full data set is divided into a collection of fixed size mini-batches. The full set of data constitute an epoch. When reading code the variable for number of mini-batches in the full set is usually called an "epoch". During training hundreds of thousands of repeated passes through the loop over "epoch" may be required for convergence of the free parameters in the models, that can number in the hundreds of millions.

CNN

CNNs create a stack of manipulations of the 2-dimensional images. The images may have multiple layers to account for colors. In such cases the color layers are manipulated independently in the first convolution in a way that their contributions are incorporated throughout the remaining network of operations.. The layers of the stack between the input and output layers are referred to as the hidden layers of the neural network. A relatively small number of APIs are all that are needed to describe most of the network and its calculations in the popular CNN structures in use today.

2-dimensional convolution

The most central is the 2-dimensional convolution. Here a small (usually square) matrix of weights and biases, the filter, are applied to the values of the image pixels and summed over to generate a single value for each filter placing. For a single layer of input the output for element m,n:

$$\text{Output}_{m,n} = \sum \text{weight}_{i+m,j+n} \text{ X } \text{input}_{i+m,j+n} + \text{bias}$$

With the sum over i,j being over the size of the filter making the convolution

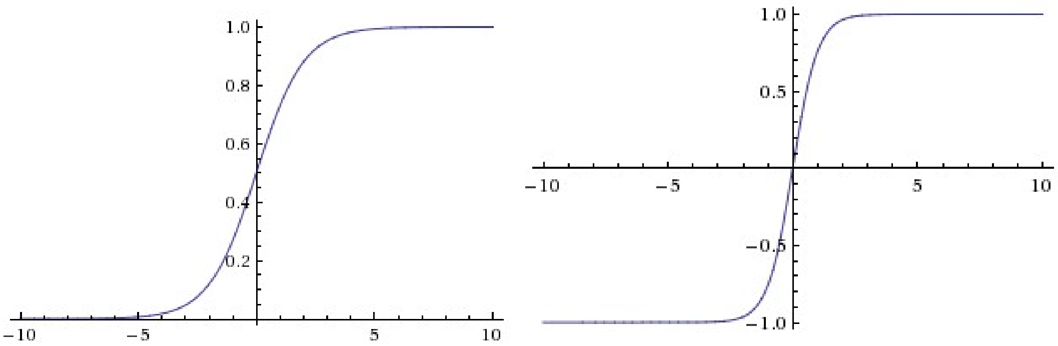
In real problems the inputs are 3 dimensional arrays and the filters are also. The operations are a convolution in 2 dimensions and a dot product in the third with one bias per input layer.

$$\text{Output}_{m,n} = \sum \sum \text{weight}_{d,i+m,j+n} \text{ X } \text{input}_{d,i+m,j+n} + \text{bias}_{d}$$

And the other sum over is over d, the full depth of input and effectively a dot product.

Activation

This output is then fed to an “activation function”. These serve the purpose of producing a sort of normalized “decision. The classic example is the sigmoid function



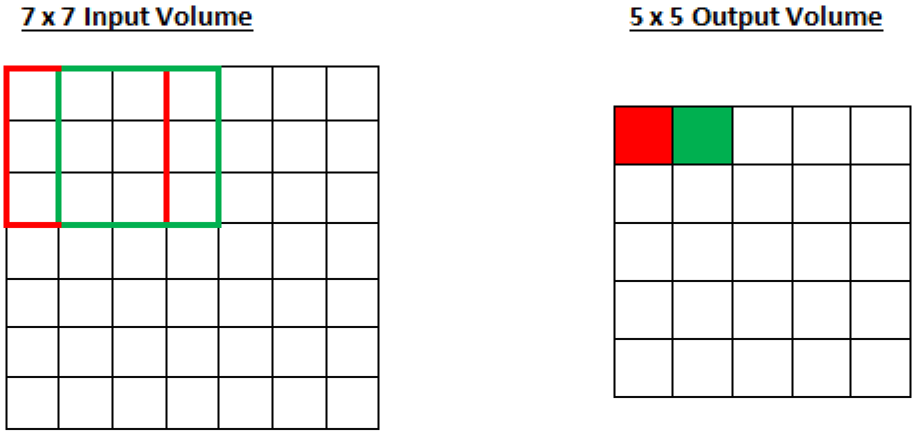
Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

Another example is the hyperbolic tangents (tanh), which can be considered a type of sigmoid. If the value being fed to the activation is a vector, the sigmoid must be generalized. The multi input version is the softmax function. In convolutional networks the most commonly used activation function is ReLu, rectified linear unit, which is simply:

ReLu(x) = max(0,x)

The sigmoid and tanh functions can produce problems with zero gradients during the learning which can make numerical management difficult. Further ReLu is much faster. It is important that the activation function can be differentiated as this is critical in the training phase (back propagation).

This combination of convolution and activation is done repeatedly, moving the filter by a stride across the surface of the image. In doing this the output image is reduced in size. In the picture below a 7X7 input array is run through a 3x3 filter with a stride of 1. This produces a 5X5 output



For this to work there is a constraint between the input dimension (W), the filter size (K) and the stride (S) so that an integer number of operations completely covers the input. To ensure this, a number of padding pixels (P) around the edges of the input may be required. This results in the output dimension (O) for the convolution being

O = 1 + (W – K +2P)/S

The same set of weights and biases are applied to each position the filter gets aligned with. The result is a series of small matrix multiplies with one of the matrixes fixed.

In building a network, the input will be processed through a number of independent filters in parallel. These blocks of filters produce a 3-dimensional block of output data. Thus the 2-dimensional convolution is one of the critical building blocks of the neural network. An example of this (Alexnet) will be discussed a bit later.

Personally, I choose to think of the filters and their trained weights and biases that make up these layers as defining a multi-dimensional space and evaluating where a particular image is in that space. In other words, each filter represents a basis vector in that space. The final layers of the network then project that space onto one defined by a finite number of tags (result options). Thus, a CNN that was trained on a very complete data set can be retrained to a new set of tagged images by only relearning the weights in the final output layers as the basis vectors remain the same.

Pooling

The next common API used in CNNs is the pooling layer. Here a small mask (usually 3X3) is swept over the 2-dimensional output array from a convolution with a stride and some operation is made for each position the mask is applied to. The most common is to take the maximum value of the input array elements covered by the mask. This is called max-pooling.

Pooling reduces the sensitivity of the result to displacements of the image. Thus, two images differing by a displacement would have close to identical results on the output of a max-pooling layer in the network.

Pooling will produce an output array that is smaller than the input array

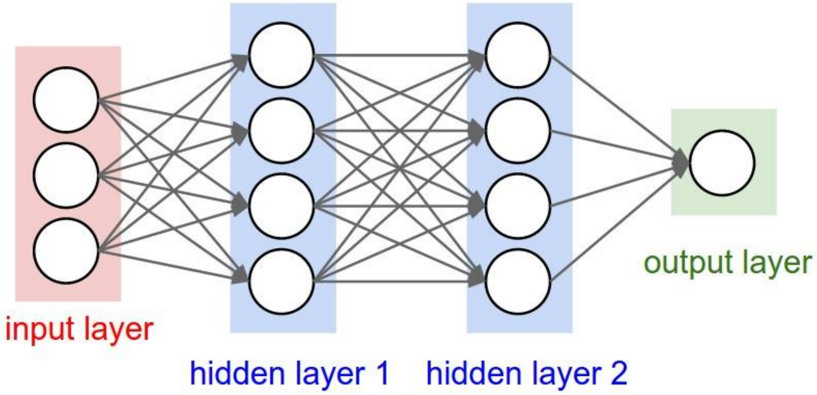
Affine or fully connected

A fully connected layer operation does the usual linear weight transformation of the inputs and then sums the outputs of all the cells for each output array element.

Output_j = ∑ weight_j_i X input_i + bias_j (sum over I, the repeated index)

The total number of weights is therefore the product of the input and output dimensions. The affine layers can introduce enormous numbers of variables (weights and biases) that must be optimized during training (aka learning)

A two layer fully connected network can be drawn as follows:



There are DNNs that consist of just fully connected layers called FCNs. The term is sometimes used in place of “fully connected layers” in CNNs and complex RNNs. Networks of FCNs are particularly useful for identifying patterns in collections of disparate data, for example identifying individuals who might be susceptible to advertisements or disinformation campaigns.

Alexnet

As illustration the code below is the function for Alexnet as encoded in the Tensorflow python framework as distributed in the TF_cnn_benchmarks suite.

```
class AlexnetModel(model.Model):
  """Alexnet cnn model."""

  def __init__(self):
    super(AlexnetModel, self).__init__('alexnet', 224 + 3, 512, 0.005)

  def add_inference(self, cnn):
    # Note: VALID requires padding the images by 3 in width and height
    cnn.conv(64, 11, 11, 4, 4, 'VALID')
    cnn.mpool(3, 3, 2, 2)
    cnn.conv(192, 5, 5)
    cnn.mpool(3, 3, 2, 2)
    cnn.conv(384, 3, 3)
    cnn.conv(384, 3, 3)
    cnn.conv(256, 3, 3)
    cnn.mpool(3, 3, 2, 2)
    cnn.reshape([-1, 256 * 6 * 6])
    cnn.affine(4096)
    cnn.dropout()
    cnn.affine(4096)
    cnn.dropout()
```

A few TF “features”: In this encoding the activation function, ReLu, was absorbed into the encoding of the conv API and so does not appear above. In addition, there is actually one more affine layer in the alexnet model. In the tf_cnn_benchmark suite this last affine layer is invoked in the function that calls AlexnetModel.add_inference.

The ‘VALID’ flag indicates that the input for the first convolution is already correctly padded and the output size can be reduced. The default value for this flag is ‘SAME’ which causes the code to pad such that the output size equals the input size during convolutions. This needs to be stated to make the size diagram (shown a bit later) make sense 😊.

Alexnet processes a 224 X 224 original image converted to 227 X 227 image, adding a border. There is a depth of 3 for RGB colors. This padding is required by the first convolution layer to make the first convolution striding fit. This issue of 224 vs 227 clearly causes some confusion in the literature 😊.

```
cnn.conv(64, 11, 11, 4, 4, 'VALID')
```

invokes 64 layers of 3 dimensional convolutions using 11X11X3 filters with strides of 4 in both dimensions. Convolutions are done as 2 dimensions of convolution + 1 dimension of dot product. In this way the input depth is absorbed, and the output depth equals the number of filters.
The output dimension of the “face” is
O =1 + (227 – 11)/4 = 55.0
Note: the API supports more than square problems.

The ‘VALID’ flag indicates that no additional padding is required.

This is fed to

```
cnn.mpool(3, 3, 2, 2)
```

which does a max pooling using a 3 X 3 mask with a stride of 2 on each of the 64 55 X 55 output arrays from the conv invocation. This produces an output array of 27 X 27
O = 1 + (55 – 3)/2 = 27

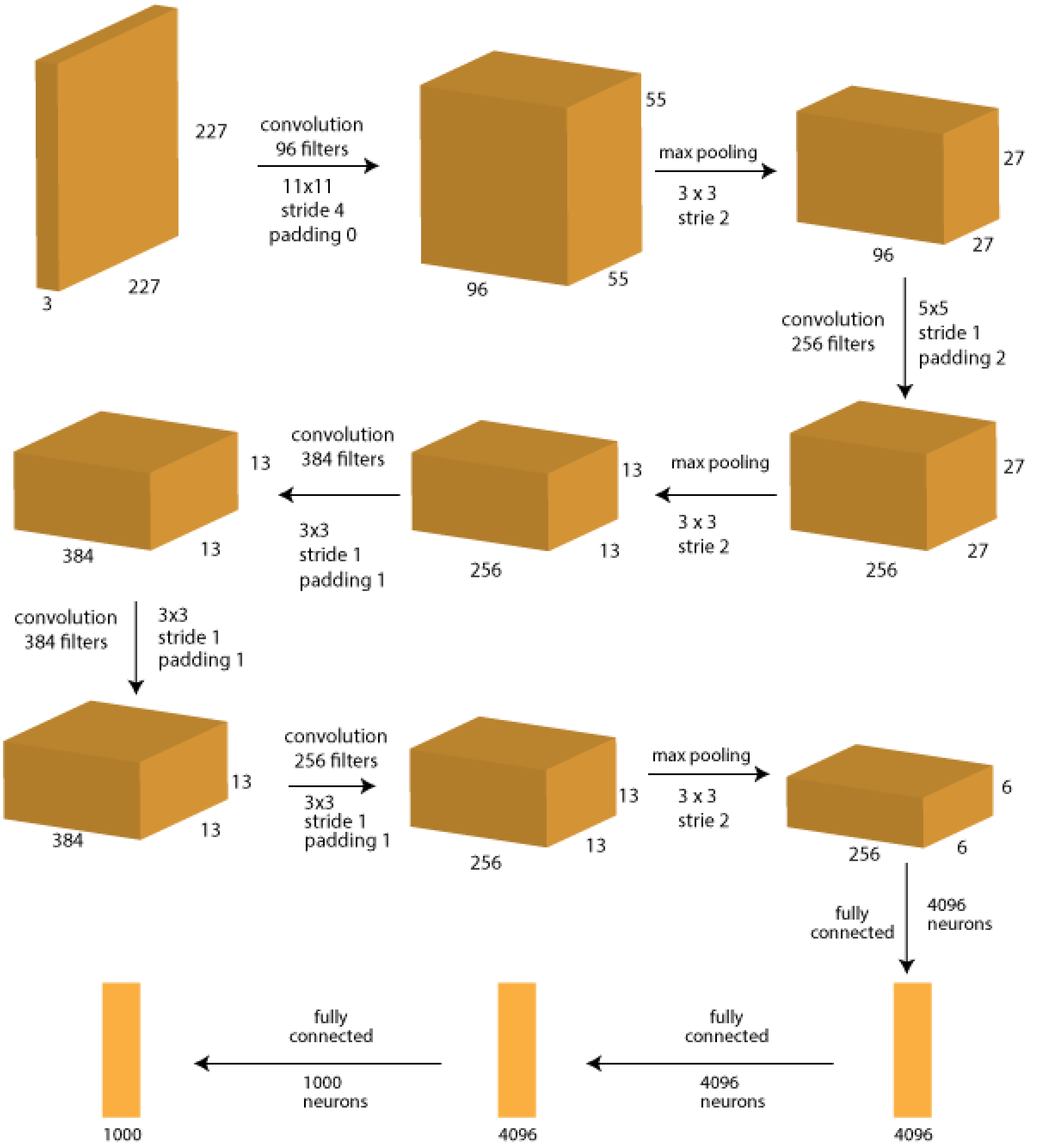
The sizes are computed in the table below. I had to deduce the required padding to ensure the output size was the ‘SAME’ as the inputs for convolutions as appropriate. The weight counts for convolutions are inputs* outputs. There is a depth of three for colors (RGB) on the first layer. A convolution layer for square filters thus requires input_depth*filter_width*filter_width*filter_depth values for weights that must be optimized during the gradient descent learning.

Note: some early versions of alexnet broke the flow into two parallel streams during certain layers. Such a dual path structure changes the calculation of the numbers of weights.
In the table below, the input depth for the first layer is 3, corresponding to the RGB layers of the image. For the subsequent layers, the input depth is the filter depth of the previous convolution.

layer	type	Input width	filter W	Padding	Stride	output	depth	weights
1	conv	227	11	0	4	55	64	23232
2	mpool	55	3	0	2	27	64	
3	conv	27	5	2	1	27	192	307200
4	mpool	27	3	0	2	13	192	
5	conv	13	3	1	1	13	384	663552
6	conv	13	3	1	1	13	384	1327104
7	conv	13	3	1	1	13	256	884736
8	mpool	13	3	0	2	6	256	

This progression is illustrated (from: https://www.analyticsvidhya.com/wp-content/uploads/2016/03/fig-8.png) below based on a slightly different alexnet model. The code above changed the numbers of filters in the first two convolutions from 96 and 256 to 64 and 192 for reasons unknown. There is a small issue in the diagram that should be pointed out. The input depth is 3 for RGB colors, and that factor of 3 gets absorbed in the first convolution and defines the depths of the filters in the first convolution.

Disclaimer: There are many networks that claim to be alexnet, they differ in the depths of the layers and many early ones appear to have 2 parallel paths that interact at various layers. Versions available that work with current frameworks tend to have single flows like the one above but may not have the same depths as the version shown here. Do not assume that the name assigned to a network uniquely identifies it and that all networks with the same “name” are equivalent.



The picture above shows the data but not the arrays for the weights and biases which occupy far more memory than the sum of the data arrays. The size of the weight arrays in the convolution layers is the filter size times the number of layers and so is not excessive. The fully connected layers need a weight for each input-output pair. Thus, the fully connected mapping from the last convolution/activation/pooling to a 4096 FCN requires 6X6X256X4096 or 38 million weights. The two other FCNs require 4096*4096 = 16.8 Million and 1000X4096 4.1 million respectively, in this case assuming the colors

Flop counts in CNNs are relatively simple to compute. They are dominated by the multiply and add

associated with the weights and biases applied to the inputs. For a convolution, the number of times a (2D) filter is applied to a given layer of the input is equal to the product of the facial output dimensions, output_width*output_width for square problems. This is done for each input depth and for each filter in the convolution. The total FP operations for the convolution layers is therefore:
FP ops = 2*Filter_width*Filter_width*Input_depth*Output_w*output_w*Output_depth
The factor of 2 is for the accumulation over the filter (assuming it is encoded as a += over a loop).

This ignores the bias terms and the sum over input layers are each output. One bias term is added to each sum of the filtered values, so the expression becomes:
FP ops = (2*Filter_width*Filter_width+1)*Input_depth*Output_w*output_w*Output_depth + Input_depth* Output_w*output_w
These extra terms are usually negligible.

Expanding the table for Alexnet to list the convolution layers, non linear activation (ReLu) and Pooling yields the following:

layer	type	Input width	filter W	output	depth	weights	FP ops	input data
1	conv	227	11	55	64	2.32E+04	1.41E+08	1.55E+05
2	ReLu	55			64		1.94E+05	1.94E+05
3	mpool	55	3	27	64		4.20E+05	1.94E+05
4	conv	27	5	27	192	3.07E+05	4.57E+08	4.67E+04
5	ReLu	27			192		1.40E+05	1.40E+05
6	mpool	27	3	13	192		2.92E+05	1.40E+05
7	conv	13	3	13	384	6.64E+05	2.37E+08	3.24E+04
8	ReLu	13			384		6.49E+04	6.49E+04
9	conv	13	3	13	384	1.33E+06	4.74E+08	6.49E+04
10	ReLu	13			384		6.49E+04	6.49E+04
11	conv	13	3	13	256	8.85E+05	3.16E+08	6.49E+04
12	ReLu	13			256		4.33E+04	4.33E+04
13	mpool	13	3	6	256		8.29E+04	4.33E+04
		input	output			weights	FP ops	
14	FCN	9216	4096			3.77E+07	7.55E+07	
15	ReLu	4096						
16	FCN	4096	4096			1.68E+07	3.36E+07	
17	ReLu	4096						
18	FCN	4096	1000			4.10E+06	8.19E+06	
19	ReLu	1000						

For the FCNs the FP ops are just 2* inputs*outputs, or twice the number of weights. Thus 76 million for the first FCN and 34 million and 8.2 million for the next two. So, while the FCNs dominate the memory requirements for Alexnet, the three FCNs produce fewer computations than just the first convolution layer by itself.

These expected values were compared against measurement on an Nvidia P4 (ubuntu 16.04, cuda toolkit 8, cudnn 6.0, TF cuda optimization set to 6.1) using NVProf. The batch size was set to 1 to minimize the optimizations invoked in the cuda libraries, but some could not be avoided. The version of alexnet used is from Soumith/Convnet (https://github.com/soumith/convnet-benchmarks) which have a different number of filters in layer 9 (table above) so the expected numbers are a little different. The results below show reasonable agreement except for the three layers of 3X3 filter convolutions.

Alexnet layer (Soumith/convnet)	expected	measured	ratio
conv (images, 3, 64, 11, 11, 4, 4, 'VALID')	140553600	143717440	1.02251
+ mpool(conv1, 3, 3, 2, 2)	small	46656	
+ conv (pool1, 64, 192, 5, 5, 1, 1, 'SAME')	447897600	472294080	1.054469
+ mpool(conv2, 3, 3, 2, 2)	small	32448	
+ conv (pool2, 192, 384, 3, 3, 1, 1, 'SAME')	224280576	162168191	0.723059
+ conv (conv3, 384, 256, 3, 3, 1, 1, 'SAME')	299040768	215558401	0.720833
+ conv (conv4, 256, 256, 3, 3, 1, 1, 'SAME')	199360512	143927547	0.721946
+ mpool(conv5, 3, 3, 2, 2)	small	9216	
+ tf.reshape(pool5, [-1, 256 * 6 * 6])	small	0	
+ affine(resh1, 256 * 6 * 6, 4096)	75497472	76988416	1.019748
+ affine(affn1, 4096, 4096)	33554432	34226176	1.02002
+ affine(affn2, 4096, 1000)	8192000	8522048	1.040289

The 3X3 convolutions can be optimized with known optimizations (Winograd, etc.) which likely explain the difference with the simple calculation. If the 3x3 convolution is replaced by a 4x4 convolution in layer 5 one would expect the number of fp operations to increase by 16/9. What one sees is:

Alexnet layer (Soumith/convnet)	expected	measured	ratio
+ conv (pool2, 192, 384, 4, 4, 1, 1, 'SAME')	398721024	453114623	1.13642

Which would seem to support the idea that the 3x3 convolution has been optimized in the cuda libraries.

If the test is run with a large mini-batch size of say 32, running NVProf will reveal that fft functions were invoked as well as cudnn functions for dense matrix operations. FFTs are convolutions so this should not be a surprise but in such cases computing the FP operation count becomes a far less obvious calculation.

The data reuse in the convolutions is a critical aspect of their execution. Each weight in the convolution layers is reused by the number of cells in an output layer, O^2.

Reuse = $(1 + (W – K +2P)/S)^2$

See http://imatge-upc.github.io/telecombcn-2016-dlcv/slides/D2L1-memory.pdf
for a discussion of memory consumption and flop counts for CNNs.

Recurrent Neural Networks (RNN)

Processing sequences of text is one of the dominant uses of machine learning in large scale usage. This was indicated by google (https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view) and has been observed elsewhere. As most data stored in data centers is text based it is easy to understand why sequence processing would be critical.

The first part of text processing that must be understood is the representation of words in a language.

Sequence inputs, word embedding and natural language processing

When parameterizing a vocabulary for the purposes on numerical processing of text a simple index list of words is not very useful. A more effective method is to assign each word a position in a multidimensional space with a position indicative of it use and relationship to other words. For example, one would like to be able to get a vector expression equality like:

Vector(France) – Vector(Paris) + Vector(Berlin) ~ Vector(Germany)

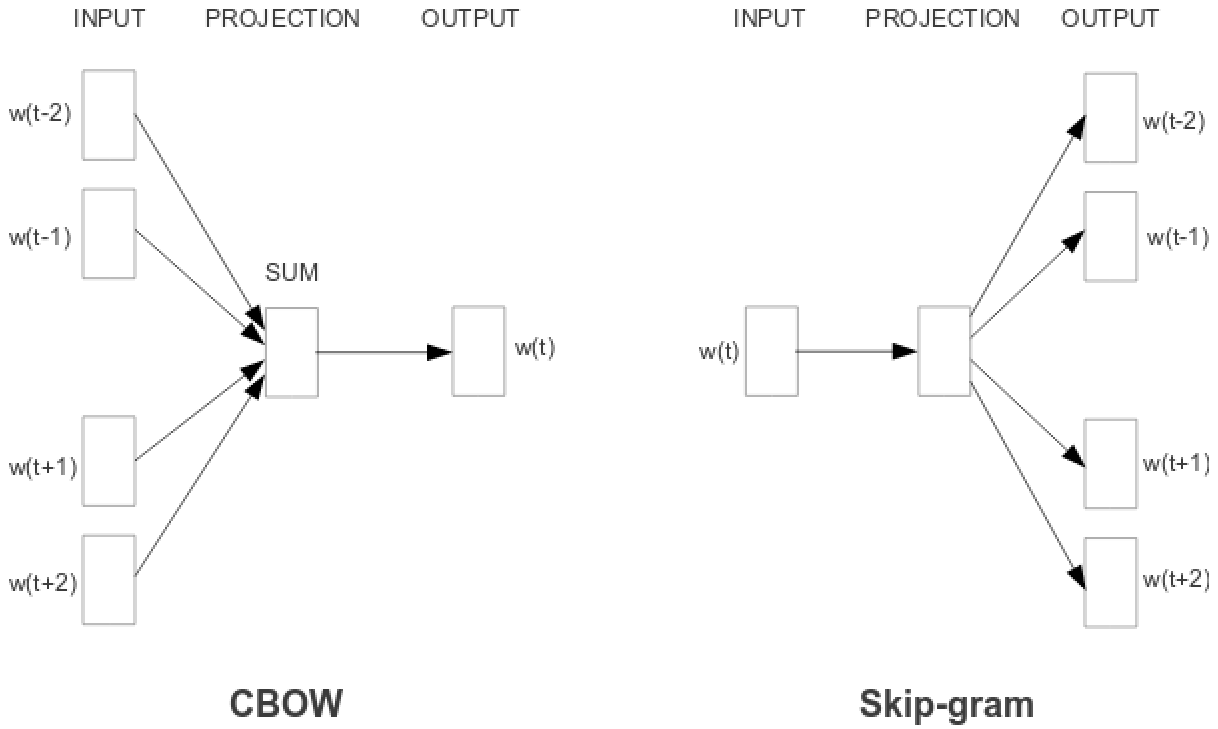
The vector evaluated on the left-hand side of the expression could be used to evaluate the cosine of the opening angle to every word in the dictionary and the largest value would be for the vector value of “germany”. Thus, something like this technique could be used to generate the desired output for sequential text processing.

The question is, how does one create a mapping of words of a vocabulary into a 2-dimensional array like

Vocabulary_embedding[vocabulary_size, embedding_dimension]

Aside: One might even like to normalize the array, so all the vocabulary vectors are normalized to unit length.

This has a long history of using various FCNs and recursive networks but ultimately a very fast and simple system was realized by Tomas Mikolov (see thesis and 1310.3781 and 1310.4546). This work resulted in the word2vec library commonly used. The library contains two related methods continuous bag of words (cbow) and skip-gram. These are basically opposite approaches to the same goal. CBOW uses the surrounding words to characterize the target, while skip-gram keys on a word and sees what it is related to.



For skip-gram the function that is maximized is the log likelihood of the target being related to the other words in the sequence (expressions from (*nlp.cs.tamu.edu/resources/wordvectors.ppt)*

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

Where

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left({v'_w}^\top v_{w_I}\right)}$$

So in practice how is this done? A training set of a large list of sentences is used for the training, which establishes the embedding vectors (ie the weights) for each word. The probabilities for each word are usually decreased with the frequency the words appear (TF-idf). Prepositions around a word tell us little about the word. So, words that are used together should get moved closer to each other.
The second part of the optimization is in realizing that randomly chosen words from the vocabulary should not be nearby. So, the target word should be pushed away from the randomly selected words.

This trick is called negative sampling or Noise Contrasted Estimation (NCE). Here the probability function used is:

$$\log \sigma({v'_{w_O}}^{\top} v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-{v'_{w_i}}^{\top} v_{w_I})\right]$$

This negative sampling technique allows much smaller samples of words to be used greatly increasing the speed of convergence. I assume that the random subsets get remixed regularly though I have not seen this stated.

The effect is that a stochastic gradient descent adjusting the weights (positions in the embedding space) will result in an embedding that reflects correlations in words and their sequences.

It should be noted that translators are trained on sets of sentences and their translations. When the embeddings for the two languages are optimized there will be a natural correlation between the words which will assist the translation.

An alternative approach is called Glove (https://nlp.stanford.edu/projects/glove/). This approach is based on first creating a co-occurrence matrix for the probability of all pairs of words, ie simply counting how many times two words occur in the same sentence divided the total occurrence rate of the first:

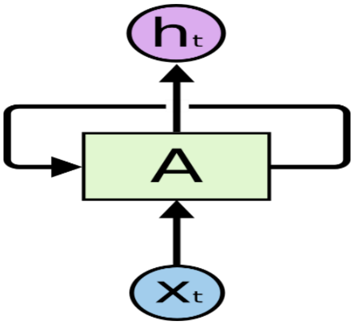
$$P_{ij} = X_{ij}/X_i$$

The resulting analysis (https://nlp.stanford.edu/pubs/glove.pdf) of this starting point produces a cost function not very different from what is ultimately reached by word2vec. It does appear to produce better correlations in word analogy tests than word2vec. The glove algorithm is used in the BiDaf code (https://allenai.github.io/bi-att-flow/) while I believe word2vec type embedding is used in NMT though there are pretrained embbedding sets based on Glove or word2vec that can be downloaded.

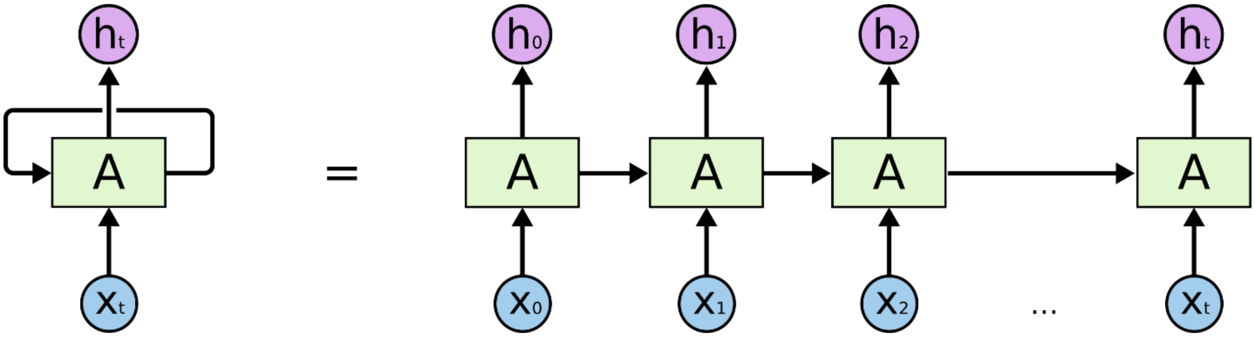
Byte Pair Encoding (BPE) is an additional step made in preparing a vocabulary before the embedding. The objective is to make subword units the vocabulary enabling the recognition of rarely used word formations from agglutination and compounding to be handled. This is done through a modification of byte pair encoding where common byte pair sets are replaced by a single unused byte. In the case of language this kind of system is built up from character sequences. Thus the words turn, turned and turning are seen to all use the root turn. (https://arxiv.org/pdf/1508.07909.pdf)

Recursive information processing

Consider reading a book. You take in the words in sequence but recall (most) of what you have already read so that the context of the incoming stream can be interpreted. This process can be drawn as a recurrent loop with a depth equal to the number of words you have read plus what you may have already known about the story. (Images below are from http://colah.github.io/posts/2015-08-Understanding-LSTMs/ which is an excellent source for further reading written by Christopher Olah)

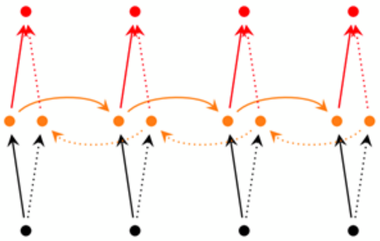


On a computer such an infinite recursion would not be practical, so the recursive loop is explicitly unrolled to a fixed width. This is referred to as the hidden width of the layer. The data passed horizontally is referred to as the hidden state



Such layers can be stacked with the outputs of one layer feeding the inputs of the next.

In the case that the text sequence or a fixed subset like a sentence is available from the beginning, as opposed to interpreting speech which intrinsically has a time ordered delivery, one could analyze the input data in reverse order just as easily. This create 2 parallel independent flows that both feed the same upper layer. Thus, each node in the upper layer has 2 inputs from the sequential stream. This can be advantageous particularly for languages that are read right to left.

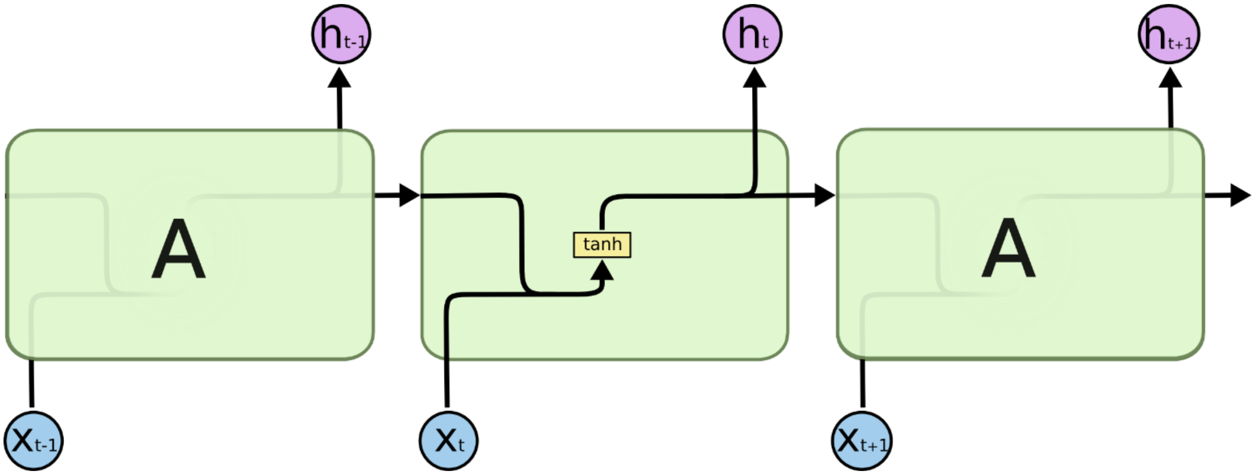


The graphic above was take from http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/, which is written by Denny Britz, who has published a great deal of useful documentation and is one of the authors of the seq2seq translator we use in the benchmark suite.

As mentioned earlier a network like either of the two preceding diagrams would qualify as a sequence to vector network. The inputs are preprocessed text, or phonemes (in the case of voice sequences) translated to embedded vectors. The last output, h_t, being the output vector.

The nodes in each layer of an RNN are referred to as cells and can become quite complex objects in themselves. The simplest cells, as shown above have 2 inputs, one from below and the "recursive" or hidden state input from the previous cell in the chain. This creates a loop carried dependency in the computations. This is one of the principal issues in achieving high functional unit utilization in the underlying architecture.

RNNs typically use the transcendental function based activations in creating the cells. The most basic rnn cell based structure (from Colah/Christopher Olah) is shown below:



And you get something like

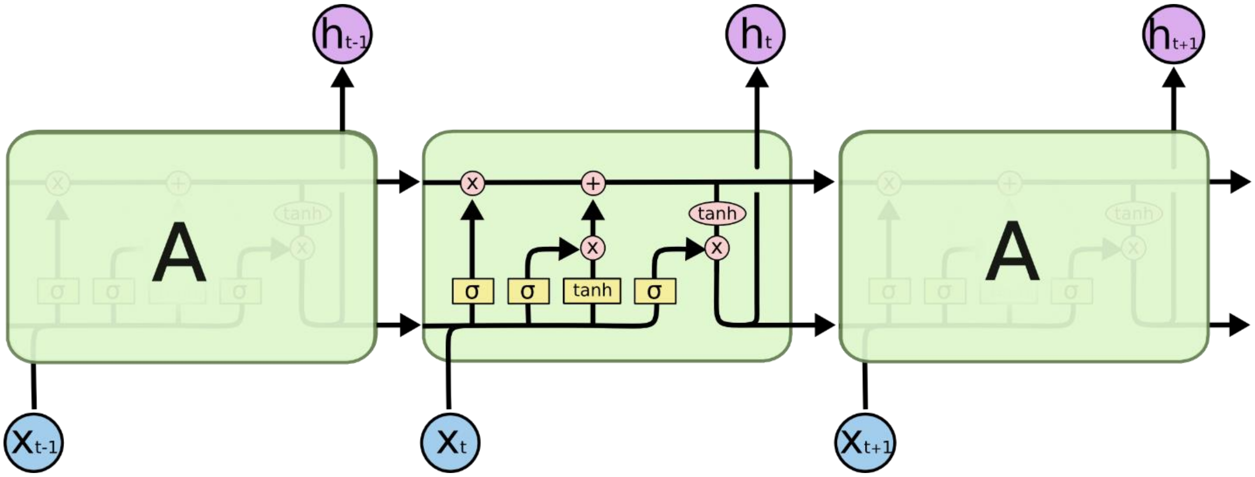
H_t = tanh(W_h*H_{t-1} + W_x*X_t + b)

This provides a long-term memory through the recursive calculation. The size of the vector H is referred to as the hidden size or in Tensorflow API documentation as the variable "num_units". The number of cells in the chain referred to as the number of steps. This can be a variable that dynamically changes to match the number of inputs or words in a sentence.
caution: the term cell in some papers refers to the hidden size. I will use cell strictly to refer to a basic unit in the RNN sequence.
In the case where a minibatch of inputs is used, there is a problem created by the variable length of the input sentences within the batch. Different frameworks handle this differently. In many of the models discussed here, during training, the sentences are ordered by size to minimize the number of idle RNN cells, as that number has to be set to the maximum in the mini batch if the framework has no built in method for handling this.

As this structure was produced from unrolling a recurrent loop the Weight matrices for a layer are the same in all the cells.

This loop carried dependency creates a problem when computing the derivatives used in the gradient descent. It produces a product of many terms and such products are numerically unstable. More will be said of this in the section on back propagation. There are various solutions to this. The most popular has been the development of the LSTM and its many variants.

It can be beneficial to have a throttling option on the long term memory component. This is provided in the Long Short Term Memory (LSTM) structure shown below (again from Colah/Christopher Olah)

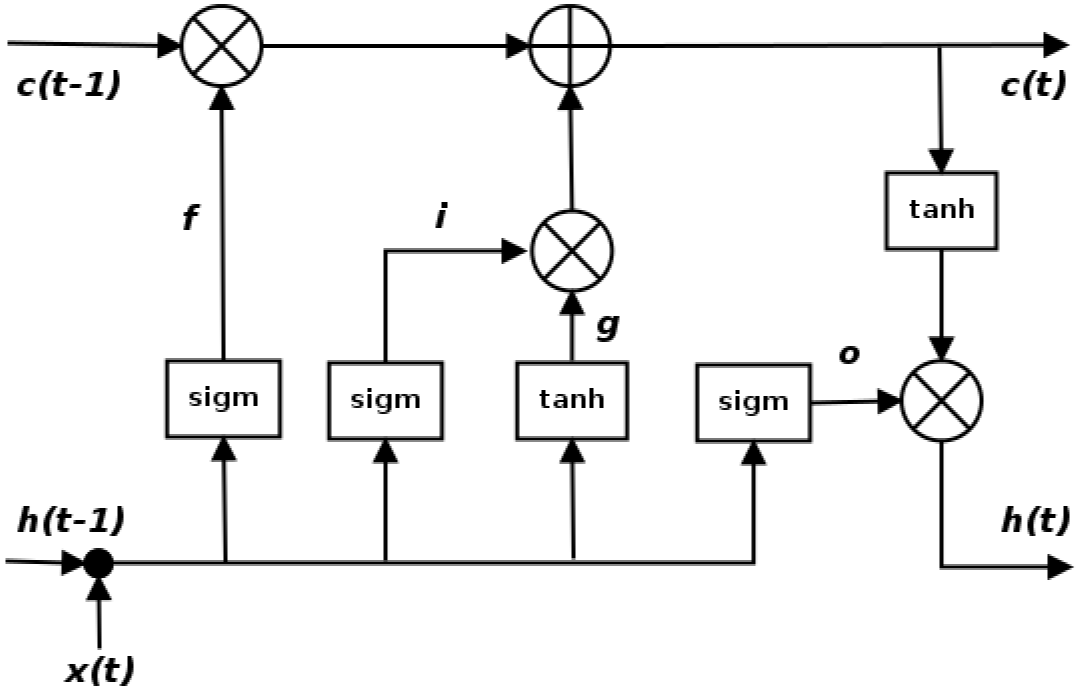


This is a considerably more complicated beast. It should be noted that there are now 2 recursive values passed horizontally. If the top hidden state value is C_t and the bottom one is S_t, one can write the math for the LSTM cell as (from Denny Britz's http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/)

$$
\begin{aligned}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \ tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{aligned}
$$

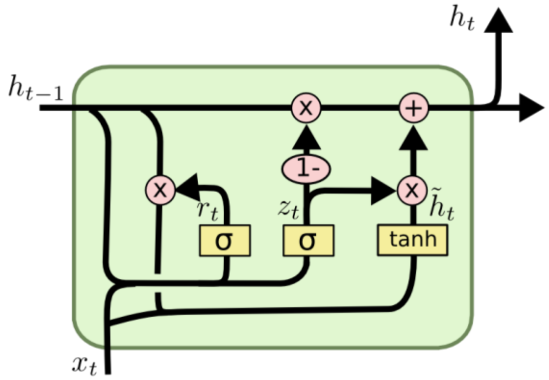
Where the U and W terms represent weights that are determined through learning.

So what is going on here? This labelled diagram (from: https://www.packtpub.com/mapt/book/big_data_and_business_intelligence/9781787128422/6/ch06lvl1sec43/long-short-term-memory--lstm) (the variable h is used here for what D. Britz referred to as "s")



The expression for g is the original term in the simple RNN cell and represents the starting point. The terms i, f and o act as gates as their activation functions are sigmoids which produce a value between 0 and 1.0. The results are then incorporated through multiplication. Thus f, the forget gate, reduces the impact of the previous steps result (c(t-1)). i, the input gate, reduces the impact of the simple RNN input value, g. The sum of the gated previous element and the gated simple RNN are the new hidden state, C_t. H_t is both the output to the next upper layer of processing and the second value of the hidden state. It is equal to C_t, the other hidden variable run through a tanh activation and then gated by o, the output gate value.

The Gated Recurrent Unit (GRU) is a variant which is somewhat simpler, though that might not be obvious from the diagram from Colah/Christoher Olah:



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

But from the math one can see the result is in fact simpler. Note there is only one variable in the horizontally passed hidden state.

Both the GRU and LSTM are standard APIs in machine learning frameworks like Tensorflow. Networks can even be coded to switch from one to the other despite the difference in the number of horizontal values passed in the hidden state.

The size of the weight arrays in the square of the hidden or embedding size of the RNN. In the LSTM each cell has 8 weight arrays while the GRU uses 6. In the NMT translator (https://github.com/tensorflow/nmt) the number of units/hidden size is set equal to the encoder and decoder embedding sizes (these could be different) and by default all are set to 1024. This ensures that all the weight matrices are square (1024 X1024)

Thus, the default 4-layer sequence to sequence model with LSTM cells, which uses 1 bidirectional layer and 3 unidirectional layers would have 5 X 1024 cells for the encoder side. The decoder side does not use bidirectional layers. Each Cell would have 8 X 1024 X 1024 weights (common within the layer) for a total of 7.5 X 10^7 weights. In a single forward pass through the RNN layers the number of FP operations is twice the number of weights times the number of cells, the sentence length, adjusted for special additions like end of sequence terms. Typical sentence lengths after additional terms and word decomposition results in 20-30 words ((https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf). Thus, a single sentence will consume something like 1.5-2 X 10^9 fp operations in the 4 layers of LSTMs.
Data blocking the calculation across cells in a layer is complicated by the large number of weights within each cell and all must be used before the weights can be reused in the next dependent cell.

If the mini batch size is incorporated directly the input sequence can become a matrix (batch size X embedding size) as can the hidden state. This enables reuse of the weights in the linear algebra. I suspect frameworks take advantage of this optimization.

The Penn Treebank benchmark (a dual layer, sequence to vector LSTM chain) sets the hidden state and number of time steps independently and explicitly tp fixed values. Looking at the code in the Tensorflow RNN tutorial (model/tutorial/rnn/ptb/ptb_lm_word.py) one finds the large model sets the hidden size to 1500 while the number of steps is 35 and there are 2 layers.

FP Operations of RNN Cells

The count of floating point operations in RNN cells during a forward pass (inference/validation) should be dominated by the vector*Matrix operations when the hidden size in large. For LSTM cells this term should be equal to 16*hidden_size*hidden_size, as there are 8 such operation. Using the Glample benchmark (https://github.com/glample/rnn-benchmarks), a single layer RNN code using synthetic data one can check this for a large range of problem sizes. The following measurements were collected with the Nvidia NVProf utility on an Nvidia P4 running TF R1.3 with python 2.7 on ubuntu 16.04:

The agreement is excellent for large hidden sizes. There is a block of 6 runs where there were twice as many FP operations as expected. This was reproducible and has not been understood.

Num Iters	Batch Size	Hidden Size	Seq Len	FP Ops	FP_ops/LSTM	expected FP Ops	Ratio
10	1	64	8	7.59E+06	94912	65536	1.45
10	1	64	16	1.52E+07	94912	65536	1.45

10	1	64	32	3.04E+07	94912	65536	1.45
10	1	128	8	2.31E+07	288128	262144	1.10
10	1	128	16	4.61E+07	288128	262144	1.10
10	1	128	32	9.22E+07	288128	262144	1.10
10	1	512	8	3.46E+08	4322816	4194304	1.03
10	1	512	16	6.92E+08	4322810	4194304	1.03
10	1	512	32	1.38E+09	4322810	4194304	1.03
10	1	1024	8	1.38E+09	17198080	16777216	1.03
10	1	1024	16	2.75E+09	17198074	16777216	1.03
10	1	1024	32	5.50E+09	17198080	16777216	1.03
10	16	64	8	9.20E+07	71872	65536	1.10
10	16	64	16	1.84E+08	71872	65536	1.10
10	16	64	32	3.68E+08	71872	65536	1.10
10	16	128	8	3.53E+08	275840	262144	1.05
10	16	128	16	7.06E+08	275840	262144	1.05
10	16	128	32	1.41E+09	275840	262144	1.05
10	16	512	8	5.49E+09	4290048	4194304	1.02
10	16	512	16	1.10E+10	4290048	4194304	1.02
10	16	512	32	2.20E+10	4290048	4194304	1.02
10	16	1024	8	2.18E+10	17067008	16777216	1.02
10	16	1024	16	4.37E+10	17067008	16777216	1.02
10	16	1024	32	8.74E+10	17067008	16777216	1.02
10	32	64	8	3.50E+08	136640	65536	2.08
10	32	64	16	7.00E+08	136640	65536	2.08
10	32	64	32	1.40E+09	136640	65536	2.08
10	32	128	8	1.37E+09	535424	262144	2.04
10	32	128	16	2.74E+09	535424	262144	2.04
10	32	128	32	5.48E+09	535424	262144	2.04
10	32	512	8	1.09E+10	4238848	4194304	1.01
10	32	512	16	2.17E+10	4238848	4194304	1.01
10	32	512	32	4.34E+10	4238848	4194304	1.01
10	32	1024	8	4.32E+10	16882688	16777216	1.01
10	32	1024	16	8.64E+10	16882688	16777216	1.01
10	32	1024	32	1.73E+11	16882688	16777216	1.01
10	64	64	8	3.63E+08	70848	65536	1.08
10	64	64	16	7.25E+08	70848	65536	1.08
10	64	64	32	1.45E+09	70848	65536	1.08
10	64	128	8	1.40E+09	272768	262144	1.04
10	64	128	16	2.79E+09	272768	262144	1.04
10	64	128	32	5.59E+09	272768	262144	1.04
10	64	512	8	2.17E+10	4238848	4194304	1.01
10	64	512	16	4.34E+10	4238848	4194304	1.01
10	64	512	32	8.68E+10	4238848	4194304	1.01

10	64	1024	8	8.64E+10	16882656	16777216	1.01
10	64	1024	16	1.73E+11	16882587	16777216	1.01
10	64	1024	32	3.46E+11	16882525	16777216	1.01
10	128	64	8	7.25E+08	70848	65536	1.08
10	128	64	16	1.45E+09	70848	65536	1.08
10	128	64	32	2.90E+09	70848	65536	1.08
10	128	128	8	2.79E+09	272768	262144	1.04
10	128	128	16	5.59E+09	272768	262144	1.04
10	128	128	32	1.12E+10	272768	262144	1.04
10	128	512	8	4.34E+10	4236776	4194304	1.01
10	128	512	16	8.68E+10	4236761	4194304	1.01
10	128	512	32	1.74E+11	4236719	4194304	1.01
10	128	1024	8	1.73E+11	16861522	16777216	1.01
10	128	1024	16	3.45E+11	16860854	16777216	1.00
10	128	1024	32	6.91E+11	16860504	16777216	1.00
10	256	64	8	1.45E+09	70848	65536	1.08
10	256	64	16	2.90E+09	70848	65536	1.08
10	256	64	32	5.80E+09	70848	65536	1.08
10	256	128	8	5.59E+09	272768	262144	1.04
10	256	128	16	1.12E+10	272768	262144	1.04
10	256	128	32	2.23E+10	272768	262144	1.04
10	256	512	8	8.68E+10	4236390	4194304	1.01
10	256	512	16	1.74E+11	4236014	4194304	1.01
10	256	512	32	3.47E+11	4236077	4194304	1.01
10	256	1024	8	3.45E+11	16858888	16777216	1.00
10	256	1024	16	6.91E+11	16858061	16777216	1.00
10	256	1024	32	1.38E+12	16858089	16777216	1.00

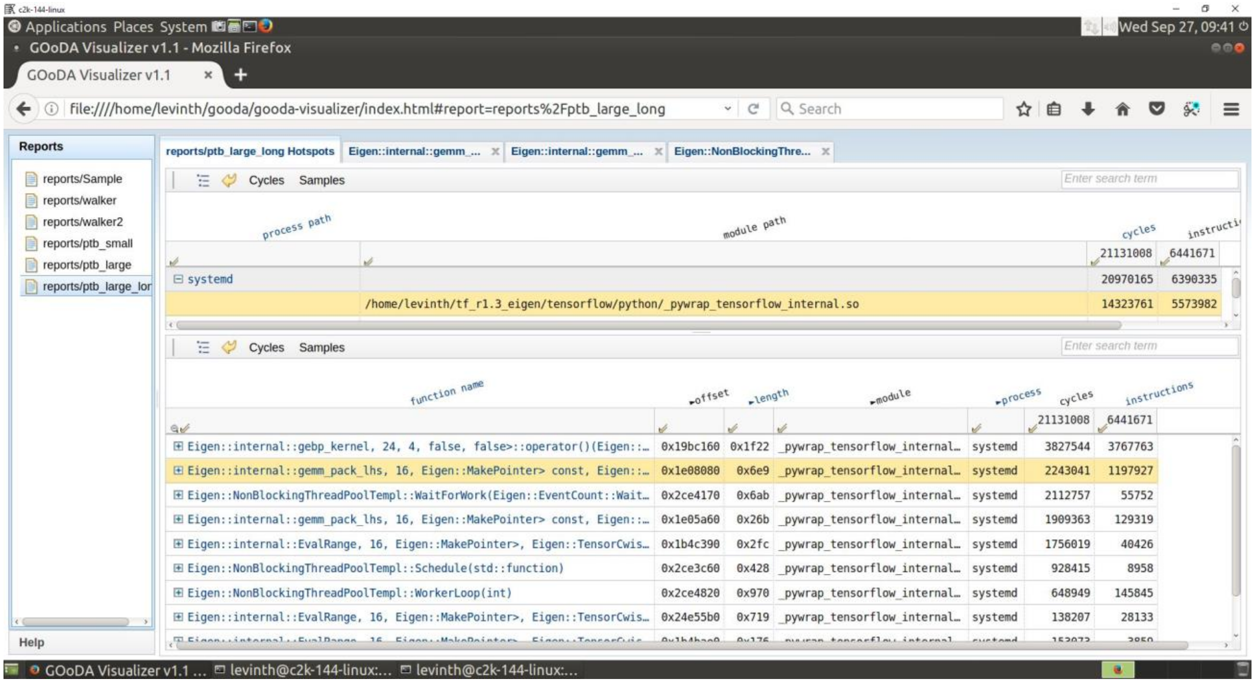
Example: Penn TreeBank (PTB)

In Tensorflow a simple single layer LSTM graph can be encoded and invoked a bit like the following code taken from the PTB tutorial (https://www.tensorflow.org/tutorials/recurrent, https://github.com/tensorflow/models Note this is from a fairly old version at about the TF_R1.0 timeframe)

```
cell = tf.contrib.rnn.BasicLSTMCell( config.hidden_size, forget_bias=0.0, state_is_tuple=True, reuse=not
is_training)
cell = tf.contrib.rnn.MultiRNNCell( [cell for _ in range(config.num_layers)], state_is_tuple=True)
self._initial_state = cell.zero_state(config.batch_size, data_type())
state = self._initial_state
outputs = []
with tf.variable_scope("RNN"):
    for time_step in range(self.num_steps):
        if time_step > 0: tf.get_variable_scope().reuse_variables()
        (cell_output, state) = cell(inputs[:, time_step, :], state)
        outputs.append(cell_output)
  output = tf.reshape(tf.concat(outputs, 1), [-1, config.hidden_size])
```

The first API defines the cell and the second defines the multi-cell network ie the number of layers. The loop then evaluates it sequentially over the number of time steps.

Running the default PTB code (tensorflow/models/tutorials/rnn/ptb/ptb_word_lm.py) on the large model with tensorflow built to use the eigen library and haswell architecture (issues building TF for avx512) and profiling the machine with perf and analyzing the perf.data file with Gooda. The data below was derived using a rather old version of the models distribution (~TF_R1.0 timeframe). (https://github.com/David-Levinthal/gooda) results in the following function breakdown of cycles and instructions:



c2k-144-linux
Applications Places System
Wed Sep 27, 09:41
GOoDA Visualizer v1.1 - Mozilla Firefox
GOoDA Visualizer v1.1
file:///home/levinth/gooda/gooda-visualizer/index.html#report=reports%2Fptb_large_long
Search
Reports
reports/Sample
reports/walker
reports/walker2
reports/ptb_small
reports/ptb_large
reports/ptb_large_lor
Help
reports/ptb_large_long Hotspots
Eigen::internal::gemm_...
Eigen::internal::gemm_...
Eigen::NonBlockingThre...
Cycles
Samples
Enter search term
GOoDA Visualizer v1.1 ...
levinth@c2k-144-linux:...
levinth@c2k-144-linux:...

process path	module path	cycles	instructi
		21131008	6441671
systemd		20970165	6390335
	/home/levinth/tf_r1.3_eigen/tensorflow/python/_pywrap_tensorflow_internal.so	14323761	5573982

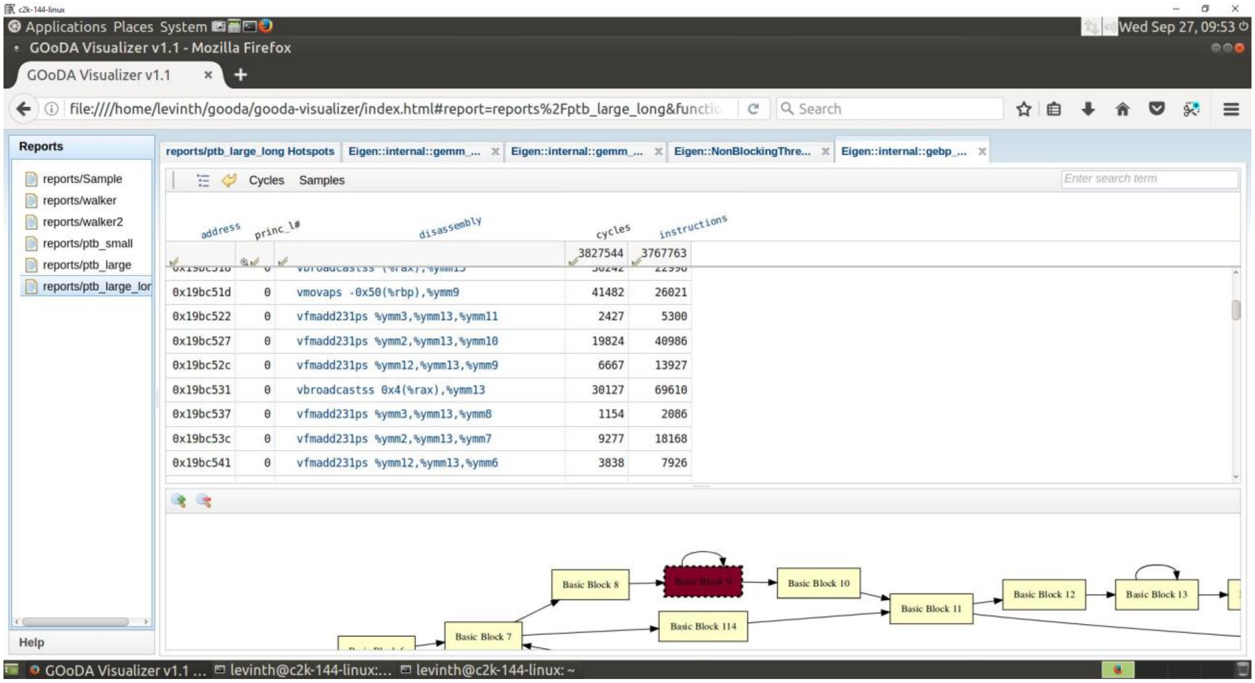
function name	offset	length	module	process	cycles	instructions
					21131008	6441671
Eigen::internal::gebp_kernel, 24, 4, false, false>::operator()(Eigen::…	0x19bc160	0x1f22	_pywrap_tensorflow_internal…	systemd	3827544	3767763
Eigen::internal::gemm_pack_lhs, 16, Eigen::MakePointer> const, Eigen::…	0x1e08080	0x6e9	_pywrap_tensorflow_internal…	systemd	2243041	1197927
Eigen::NonBlockingThreadPoolTempl::WaitForWork(Eigen::EventCount::Wait…	0x2ce4170	0x6ab	_pywrap_tensorflow_internal…	systemd	2112757	55752
Eigen::internal::gemm_pack_lhs, 16, Eigen::MakePointer> const, Eigen::…	0x1e05a60	0x26b	_pywrap_tensorflow_internal…	systemd	1909363	129319
Eigen::internal::EvalRange, 16, Eigen::MakePointer>, Eigen::TensorCwis…	0x1b4c390	0x2fc	_pywrap_tensorflow_internal…	systemd	1756019	40426
Eigen::NonBlockingThreadPoolTempl::Schedule(std::function)	0x2ce3c60	0x428	_pywrap_tensorflow_internal…	systemd	928415	8958
Eigen::NonBlockingThreadPoolTempl::WorkerLoop(int)	0x2ce4820	0x970	_pywrap_tensorflow_internal…	systemd	648949	145845
Eigen::internal::EvalRange, 16, Eigen::MakePointer>, Eigen::TensorCwis…	0x24e55b0	0x719	_pywrap_tensorflow_internal…	systemd	138207	28133

Or:

function	module	"cycles"	"instructions"
Eigen::internal::gebp_kernel<float, float, long, Eigen	"_pywrap_tensorflow_internal.so"	3827544	3767763
Eigen::internal::gemm_pack_lhs<float, long, Eigen::i	"_pywrap_tensorflow_internal.so"	2243041	1197927
Eigen::NonBlockingThreadPoolTempl<tensorflow::tl	"_pywrap_tensorflow_internal.so"	2112757	55752
Eigen::internal::gemm_pack_lhs<float, long, Eigen::i	"_pywrap_tensorflow_internal.so"	1909363	129319
Eigen::internal::EvalRange<Eigen::TensorEvaluator<	"_pywrap_tensorflow_internal.so"	1756019	40426
Eigen::NonBlockingThreadPoolTempl<tensorflow::tl	"_pywrap_tensorflow_internal.so"	928415	8958
Eigen::NonBlockingThreadPoolTempl<tensorflow::tl	"_pywrap_tensorflow_internal.so"	648949	145845
Eigen::internal::EvalRange<Eigen::TensorEvaluator<	"_pywrap_tensorflow_internal.so"	138207	28133
Eigen::internal::EvalRange<Eigen::TensorEvaluator<	"_pywrap_tensorflow_internal.so"	153073	3859
std::_Function_handler<void (long, long), Eigen::inte	"_pywrap_tensorflow_internal.so"	101481	5729

GEBP is an encoding of the Goto, Van de Geijn BLAS L3 kernel (http://www.cs.utexas.edu/users/flame/pubs/flawn20.pdf), which is what we might expect given that our calculations indicate that the computations are dominated by matrix vector multiplies

The disassembly and CFG can be displayed in Gooda to show it being dominated by vfmadd231ps instructions:


c2k-144-linux
Applications Places System
Wed Sep 27, 09:53
GOoDA Visualizer v1.1 - Mozilla Firefox
GOoDA Visualizer v1.1
file:///home/levinth/gooda/gooda-visualizer/index.html#report=reports%2Fptb_large_long&functio
Search
Reports
reports/Sample
reports/walker
reports/walker2
reports/ptb_small
reports/ptb_large
reports/ptb_large_lor
Help
reports/ptb_large_long Hotspots
Eigen::internal::gemm_...
Eigen::internal::gemm_...
Eigen::NonBlockingThre...
Eigen::internal::gebp_...
Cycles
Samples
Enter search term
Basic Block 8
Basic Block 9
Basic Block 10
Basic Block 11
Basic Block 12
Basic Block 13
Basic Block 7
Basic Block 114
GOoDA Visualizer v1.1 ...
levinth@c2k-144-linux:...
levinth@c2k-144-linux: ~

```
address    princ_l#  disassembly                          cycles   instructions
                                                          3827544  3767763
0x19bc51d  0         vmovaps -0x50(%rbp),%ymm9            41482    26021
0x19bc522  0         vfmadd231ps %ymm3,%ymm13,%ymm11      2427     5300
0x19bc527  0         vfmadd231ps %ymm2,%ymm13,%ymm10      19824    40986
0x19bc52c  0         vfmadd231ps %ymm12,%ymm13,%ymm9      6667     13927
0x19bc531  0         vbroadcastss 0x4(%rax),%ymm13        30127    69610
0x19bc537  0         vfmadd231ps %ymm3,%ymm13,%ymm8       1154     2086
0x19bc53c  0         vfmadd231ps %ymm2,%ymm13,%ymm7       9277     18168
0x19bc541  0         vfmadd231ps %ymm12,%ymm13,%ymm6      3838     7926
```

One can use the PTB test to check the FP operations for an LSTM cell. It was observed that there is a considerable baseline for a single LSTM layer in the TF PTB tutorial example. Consequently, the slope of the PTB FP operations as layers are added to the PTB model is used to check the FP operations/LSTM structure. As stated earlier this is expected to be 8*2*hidden_size*hidden_size.

Adjusting the model so that

hidden_size	minibatch size	LSTM seq len
1024	128	32

This results in there being 17 mini batches in the data set (epoch variable in code =17). The hidden size of 1024 would suggest that each LSTM construct should result in 16777216 FP operations for the calculations of the input vectors times the 8 weight matrices.

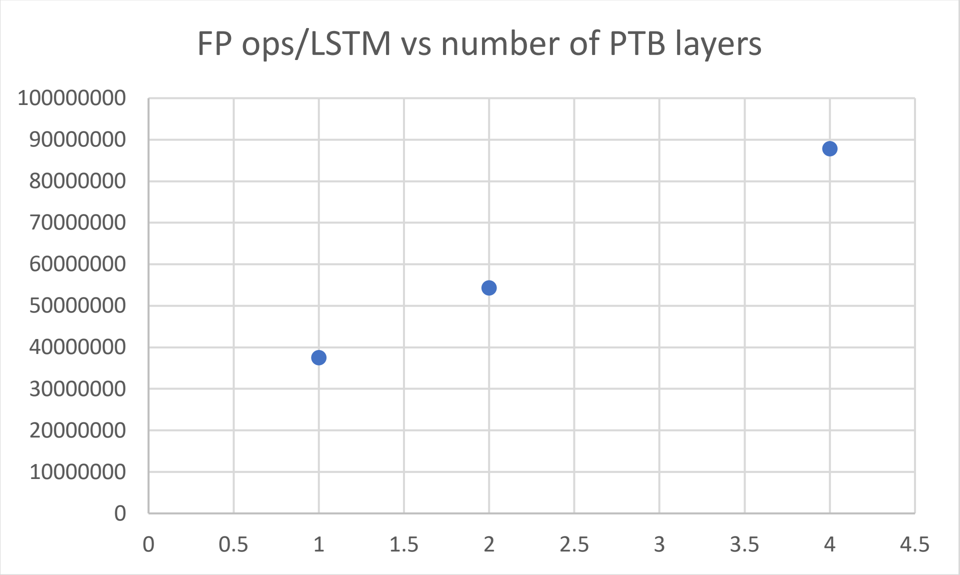
Using Nvidia NVProf tool on a P4 while running the code on TF 1.3 (python 2.7) we can compare the total number of FP operations in the forward pass to the FP operations only in the dense matrix code, sgemm:

	1 Layer	2 Layers	4 Layers
total fp_ops	2.64E+12	3.82E+12	6.16E+12
Sgemm fp_ops	2.61E+12	3.78E+12	6.12E+12

Clearly the matrix operation dominates the FP operation count. Using only the Sgemm counts we can compare the results to the expectation:

	sgemm fp ops	sgemm fp opps/LSTM	2 point slope	ratio slope/expected value
1 layer	2.61E+12	3.75E+07		
2 layers	3.78E+12	5.43E+07	16781312	1.000244
4 layers	6.12E+12	8.78E+07	16781312	1.000244

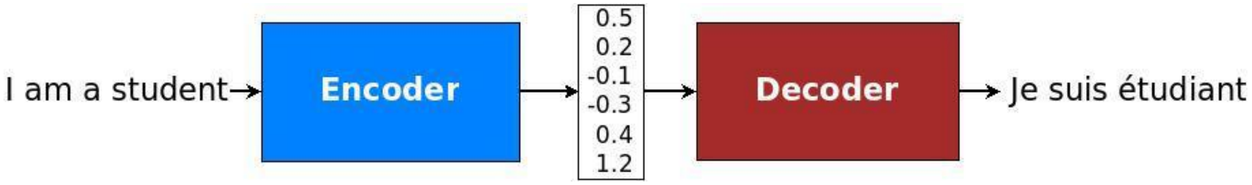
Clearly there is excellent agreement.



Knowing that we have a solid method for measuring the FP operations during a forward pass through an LSTM cell suggests that it should be possible to measure the FP operations for back propagation. This will be discussed in the section on back propagation.

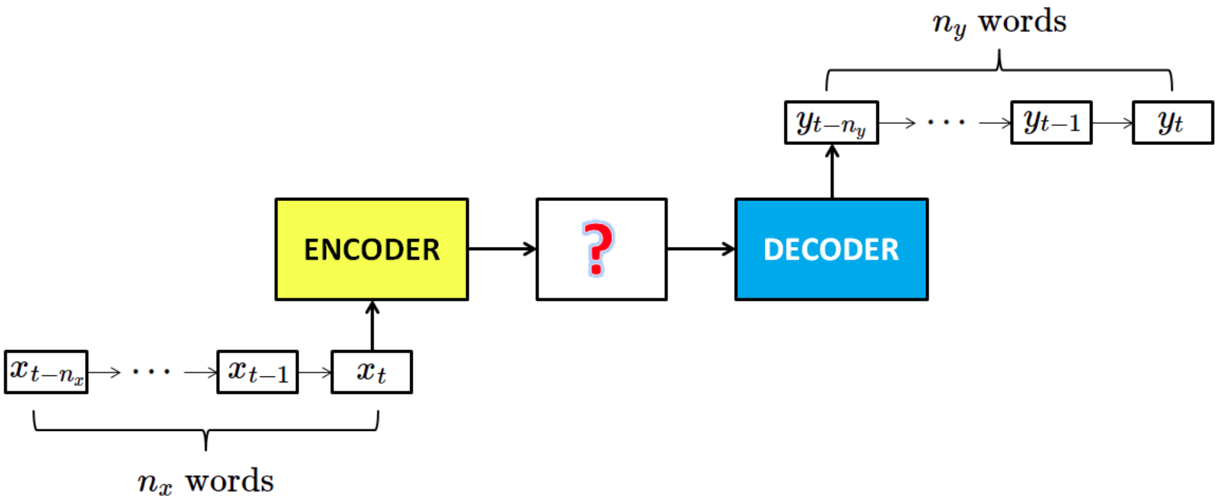
Sequence to Sequence or seq2seq

The sequence to sequence structure was introduced as a mechanism to handle transformations of input sequences to output sequences of different length (https://arxiv.org/pdf/1409.3215.pdf). The input sequence is processed through an input RNN structure called the encoder producing a fixed length vector or context as an intermediate representation of the entire sequence. This is fed to another RNN structure which then decodes the context into the desired output. The structure is well suited for language translators, or other types of sequence transformations (input words to phonems, etc). The following diagram illustrates the concept (take from https://github.com/tensorflow/nmt)

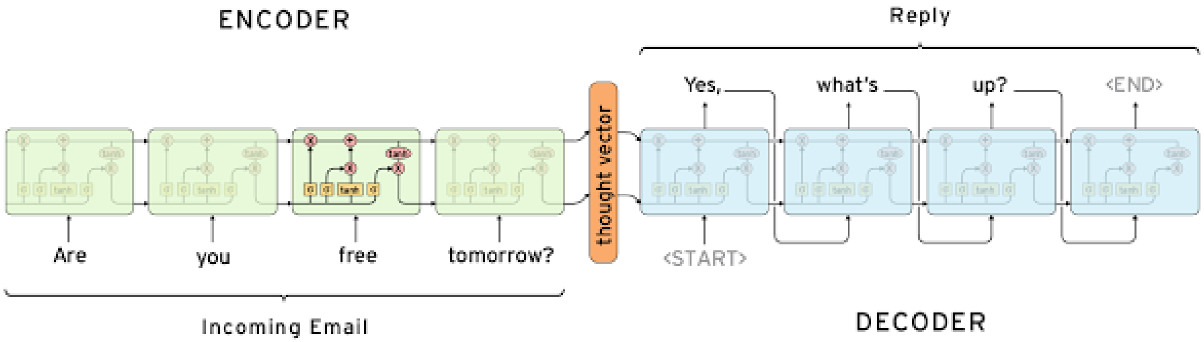


The final output of the encoder is a context, which is passed as a stream to the decoder. This decoupling allows the output length to be independent of the input length. (diagram from (https://chunml.github.io/ChunML.github.io/project/Sequence-To-Sequence/). It requires that all information about the input sequence can be completely encoded into the hidden state of the final output RNN cell (simple RNN, LSTM or GRU or a variant). This can be a limitation for long input sequences as the size of the hidden state is statically set on most frameworks.

When a seq2seq configuration is used for language translation it should be kept in mind that the word embedding can be trained of identical data in both languages creating a correlation in the embedded encoding of words in the two languages.



A second form of the diagram (for a chatbot) from Denny Britz's http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/ is shown below with the RNN's LSTM cells displayed.



In the original seq2seq translator paper (https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf) the context vector was 1000 fp values. The paper states that 32M values are used in the encoder and decoder weights for 4-layer LSTM networks. This is 1 million weights/weight matrix which is what is expected for 1000 element vectors.
In such a system the embedded word size might not be the same size as the hidden state resulting non square matrices for the first layer. This is frequently the case in practice, but is not required, as can be seen with the sockeye Neural Machine Translation application (MXNET based, https://github.com/awslabs/sockeye) where the training example offered in the WMT tutorial uses an embedded size of 256 and a hidden size of 512. Similarly, the Nematus translator (Theano based, https://github.com/EdinburghNLP/nematus) can also support different sizes for these parameters.

It might also be connected such that the context vector feeds all the output/decoder cells (figure from "Deep Learning" by Goodfellow et al) as one of the hidden state vectors and kept as a constant.

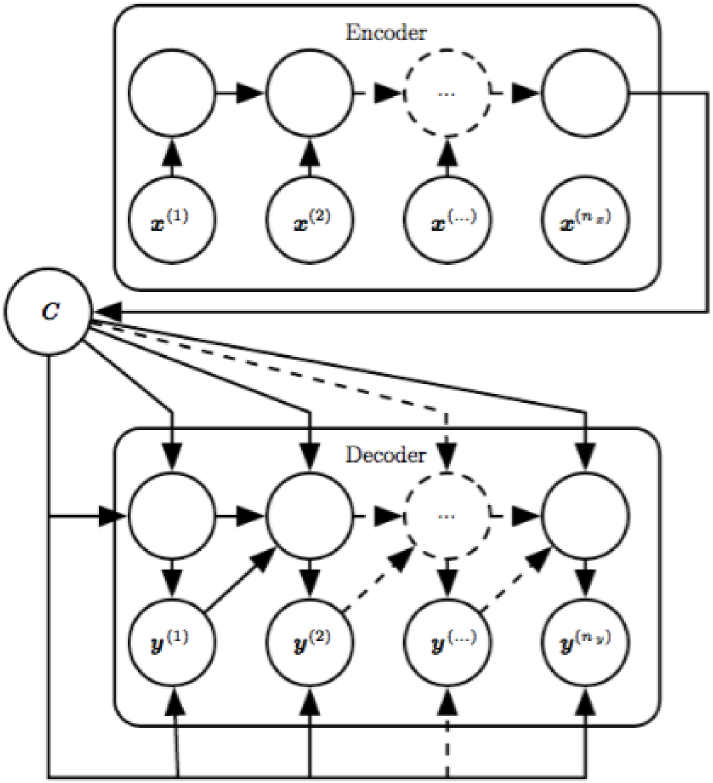


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)})$ given an input sequence $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n_x)})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

Output generation and beam search

The final word selection for the output of a translator is generated from another vocabulary with a similarly trained embedding vector structure. In the original seq2seq translator of Sutkever et al (https://arxiv.org/pdf/1409.3215.pdf) the stream of the last several translated words and the N most probable output sequences to that point (the beam search) was used to determine the best correlation for the next word looping over the entire target vocabulary. Such a search over an entire vocabulary and the use of a softmax weighting can create the weights for each term hypothesis as they are added. As only the sequences with the highest total probability (product of all term probabilities to that point) are propagated through the search. (https://www.youtube.com/watch?v=UXW6Cs82UKo)

Softmax use in RNNs

The softmax function is a normalized exponentially weighting of the components of a vector

$$S(x_t) = \exp(f(x_t))/\sum_i \exp(f(x_i))$$

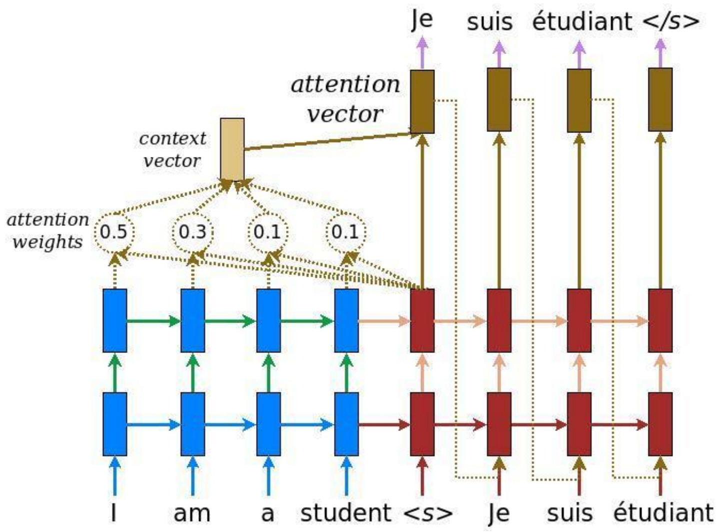
Where the function f can be a wide variety of possibilities. The softmax function peaks for the largest values in the vector so this sort of weighting is dominated by the major components. This makes it a very useful function for identifying which components should propagate further. It also has the property that it is differentiable, greatly helping backpropagation. Using a “argmax” type of function would make this difficult. When the vectors become large (example a vocabulary) this can become a difficult function to compute.

Attention mechanism

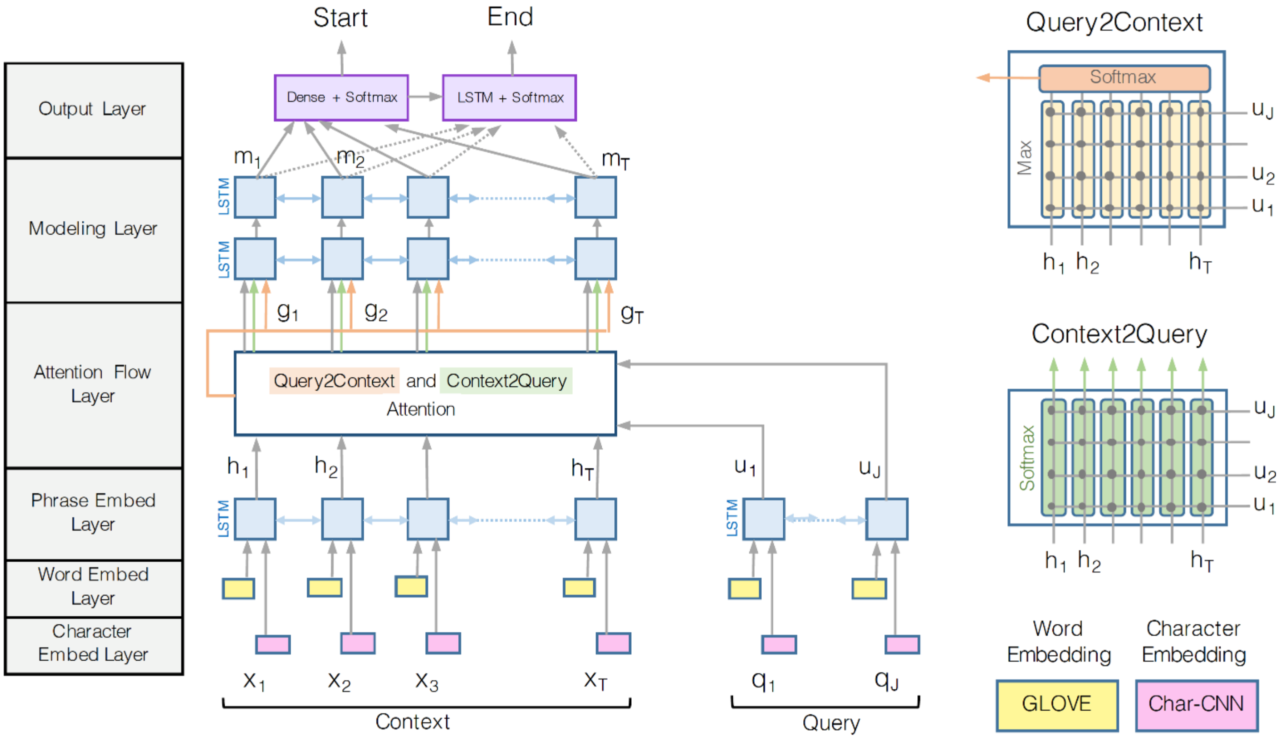
One problem with RNNs is that when the input sequences get extremely long the interpretation of the context can degrade as it cannot adequately encode all the information about the input sequence. This is shown by the BLEU scores for translators decreasing for long sentences (fig 6 of1409.0473 illustrates this). A mechanism for alleviating this, called “attention” (https://arxiv.org/pdf/1409.0473.pdf), was introduced recently. The idea is that all the output vectors/hidden state from the top layer (H_t) of the encoding RNN can be dynamically combined in a weighted sum driven by the state of the decoder loop as that loop progresses. This is driven by the observed relationship between a target word and the words of the input sentence. This relationship trains the weights in the attention component of the network. This in turn then drives the final output.

A way to think about this is that the current state of the output can provide insight into which components of the input are most relevant. In the case of question and answer RNN systems like BiDaF (https://allenai.github.io/bi-att-flow/) the source text and question must be combined to generate meaning for the question and a sequence of context vectors for the answer generation.

(picture from nmt/Luong readme)



A question-answer seq2seq with attention would be constructed as follows (diagram from https://allenai.github.io/bi-att-flow/):



I believe that in NMT, as each successive output is generated it is used as an input to the attention mechanism. This is illustrated dynamically in a diagram in https://distill.pub/2016/augmented-rnns/ from Olah and carter (Olah & Carter, "Attention and Augmented Recurrent Neural Networks", Distill, 2016.). I cannot incorporate that dynamic display into this document. The objective is to focus the context vector through a weighted sum of the hidden states from the encoder based on the current state of the output (ie which output word has been reached).

There are a rather wide variety of algorithms in use to generate a time (sequence position) dependency on the context vector passed to the decoder. From basic softmax based methods to FCNs to generate the correlation weights for combining the sequence of hidden state vectors from the top of the encoder. A score is created for each time step of the encoder and a weighted sum of the hidden states is then used for the context vector.
The paper of Luong et al (https://arxiv.org/pdf/1508.04025.pdf) discusses using multiple attention mechanisms in neural machine translation.

$$\mathbf{W_t(S) = exp\,(score\,(D_c\,,S_t)\,/\textstyle\sum_i exp\,(score\,(D_c\,,S_i)\,)}$$

Where D_c is the current decoder hidden state and S_i is the encoder’s hidden state for time step i.
Consider the following 3 options for the scoring function (luong). (h_t is the current decoder hidden state and h_s is one of the top layer encoder hidden states

$$\mathrm{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

1) A simple dot product of the decoder hidden state and the top layer hidden state for a given time step. What happens here is that the word embedding scheme will through its training identify the correlation even across a translation
2) A general learned weighting between the vector components of the decoder and encoder
3) Concatenating the encoder and decoder vectors (producing a vector of twice the length) and then applying a learned weight followed by the tanh on each component and then a dot product with a learned vector.

The diagram below (from http://colinraffel.com/publications/iclr2016feed.pdf) illustrates using a single layer FCN which is optimized during training. This would correspond to the second choice in the list above.

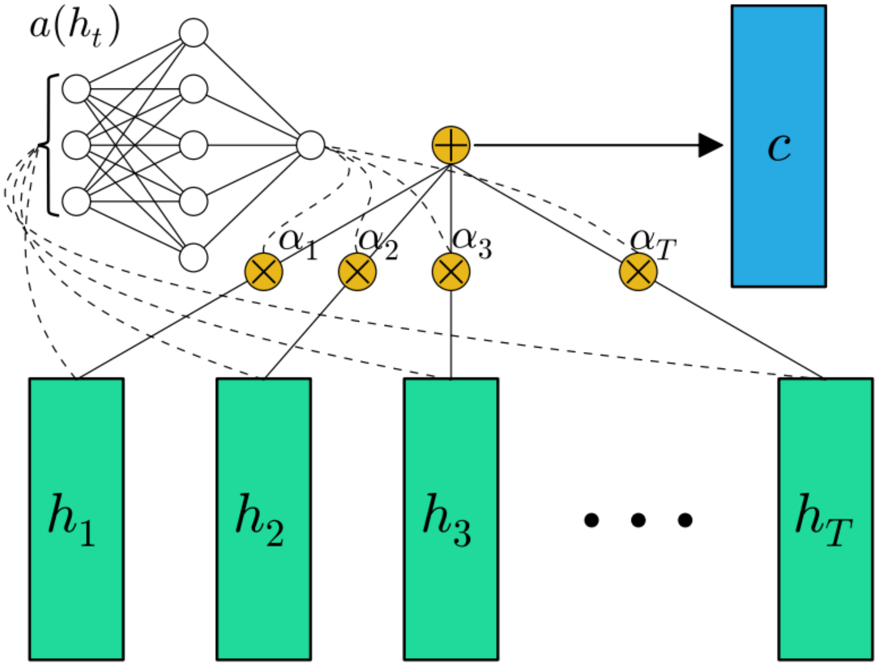
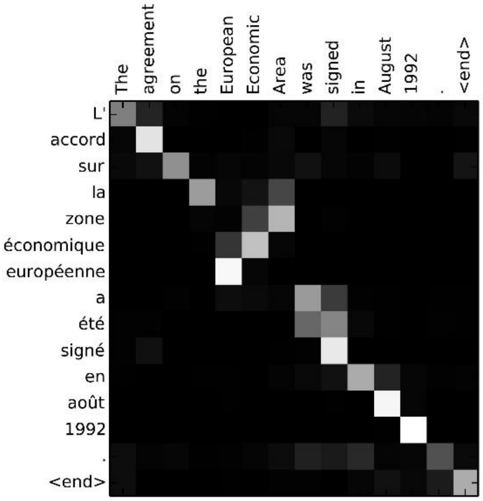
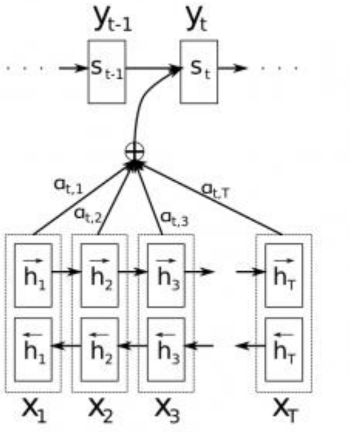


Figure 1: Schematic of our proposed "feed-forward" attention mechanism (cf. (Cho, 2015) Figure 1). Vectors in the hidden state sequence h_t are fed into the learnable function $a(h_t)$ to produce a probability vector α. The vector c is computed as a weighted average of h_t, with weighting given by α.

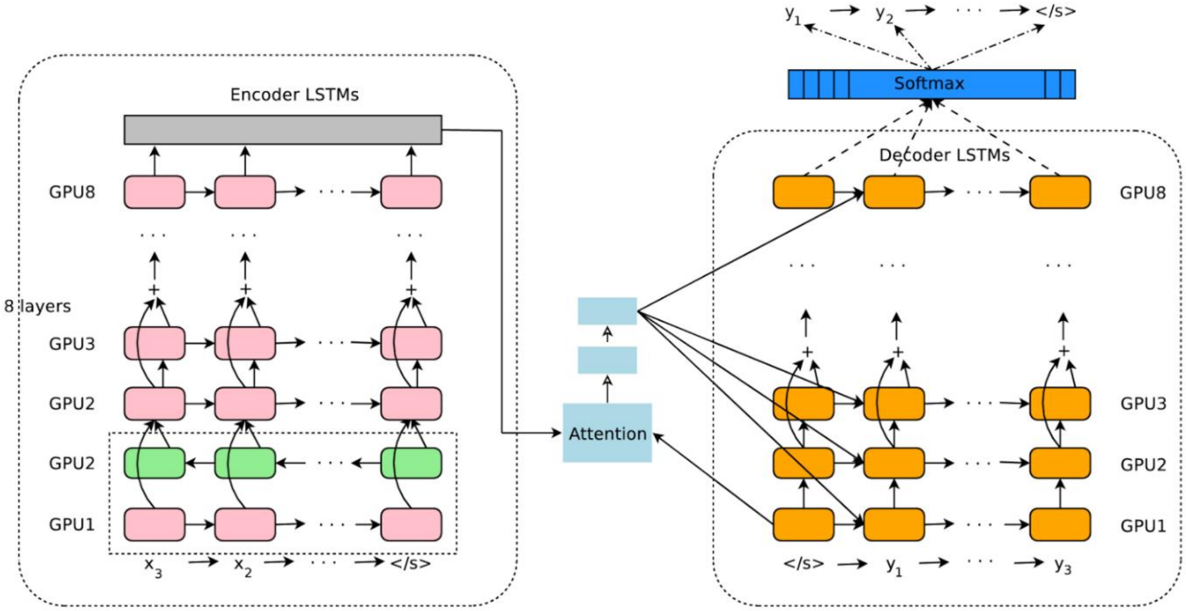
The result is to focus the locality of the inputs. (from the original Bahdanau et al paper)



From: https://arxiv.org/pdf/1409.0473.pdf (Original Attention paper by Bahdanau et al)



When put together with the 8 layer seq2seq language translator (NMT) one has a network like the following from https://sites.google.com/view/seq2seq-icml17



Backpropagation

Backpropagation is really the magic that makes ML possible. A chain rule analysis of the dependence of the outputs on an individual weight or bias reveals that it can be reduced to a recursive calculation backwards through the network. This makes the calculation local to the node and the nodes in the next (higher) layer.

Following (https://sudeepraja.github.io/Neural/) one finds that the derivative of the error of the result on a weight of a given layer depends only on terms for the next higher layer and the activation function of the given node. Thus, the derivatives can be calculated iteratively walking backwards through the layers. For the example a quadratic error (χ^2) leads to the following expressions:

Where $f_i(W_i x_{i-1})$ is the activation of a node in the ith layer (and ignoring biases) and f' is the derivative of the activation wrt the weight.

Forward Pass:

$$x_i = f_i(W_i x_{i-1})$$
$$E = 1/2 \| x_L - t \|^2$$

Backward Pass:

$$\delta_L = (x_L - t) \circ f'_L(W_L x_{L-1})$$
$$\delta_i = W^T{}_{i+1} \delta_{i+1} \circ f'_i(W_i x_{i-1})$$

The Weight update goes in the opposite direction of the derivative to reduce the error:

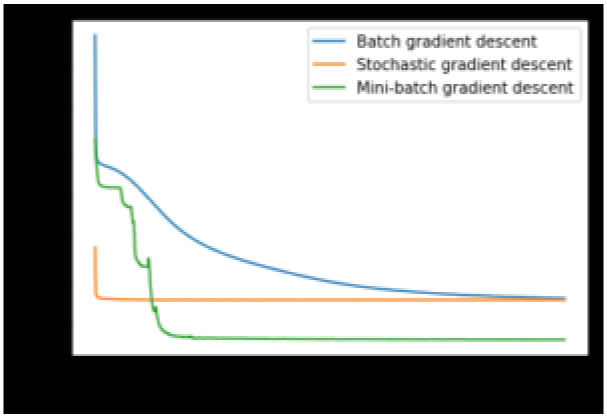
$$\partial E / \partial W_i = \delta_i x^T{}_{i-1}$$
$$W_i = W_i - \alpha_{Wi} \circ \partial E / \partial Wi$$

Where α_{Wi} is the learning rate for W_i

Evaluating and changing the weights with every input sample in the training data set raises the cost of training and makes all the fluctuations manifest. Further there is the possibility of this leading to finding a local (or false) minimum and not reaching the best possible result.

What is done in practice is to break the training data set into mini batches and then update the weights with the average of the changes over the mini batch. This helps stabilize the stochastic gradient descent. Further in CNNs, as the weights are heavily reused, there are many terms for the change in the weights when backpropagation is evaluated. Once again these are averaged, further stabilizing the optimization.

The graph below illustrates how stochastic gradient descent (ie updating weight after each sample) reaches a minimum quickly but does not reach the true minimum. It also illustrates that updating the weights once per full data set (batch gradient descent) takes longer and can find the wrong minimum while the added numerical stability of using mini batches and adjusting the learning rate (down) from time to time allows the false local minima to be avoided. (graph and eqs below from: http://adventuresinmachinelearning.com/stochastic-gradient-descent/)



The equations about are modified for the averaging along the lines of (J is the cost function and the gradient is calculated wrt the weights) as above:

$$W=W-\alpha\nabla J(W,b,x_{(z:z+bs)},y_{(z:z+bs)})$$

Where bs is the mini-batch size and the cost function is:

$$J(W,b,x_{(z:z+bs)},y_{(z:z+bs)})=1/bs\sum_{z}J(W,b,x_{(z)},y_{(z)})$$

In the case of RNNs there is also the derivative through the sequence due to the hidden state, but the issue is now complicated as the weights are the same within the layer. The result of this is that the derivative of the output on the first input cell ends up with a term of the weight matrix raised to the number of steps minus 1. To see this, consider a simple network with no nonlinear activations and

$$h_t = W_h * h_{t-1} + W_x * X_t$$

The derivative back propagated through the layer will produce a large power of the weight matrices. Thus, if any eigen values become substantially less than 1 the product vanishes making the whole thing numerically unstable.

Floating point operations for back propagation

It is possible to measure the number of floating point operations associated with back propagation using the techniques described earlier. In this case One applies the same technique but then takes the difference of the training result and the forward pass/verification/inference result. The PTB benchmark was run as follows (ie same hidden size, number of mini batches, seq length for training and verification)

hidden_size	minibatch size	LSTM seq len	verification epoch	training epoch
1024	128	32	17 mini batches	226 mini batches

With results as follows:

	1 layer	2 layers	4 layers
verification	2.61126E+12	3.77977E+12	6.1168E+12
training	1.04537E+14	1.51361E+14	2.45011E+14
verification-base)/LSTM		16781312	16781312
verification-base)/LSTM		50582310	50583055
back propagation/LSTM		33800998	33801743

Thus, it we conclude that training uses ~ 3 times as many FP operations as inference or that the backpropagation requires ~twice as many FP operations as the forward pass. The throughput, as measured in words per second, was 4.4 times higher for the verification (large batch size inference) than for the training on this same hardware/software configuration.

Acknowledgements:

I would like to thank all the authors of the books, blogs and articles listed below, besides those explicitly mentioned in the text. Any errors in the test are purely mine and anything correctly explained is owed to all of them. I would particularly like to thank my colleague, Rajiv Kapoor, for many useful discussions and assistance with the data collection.

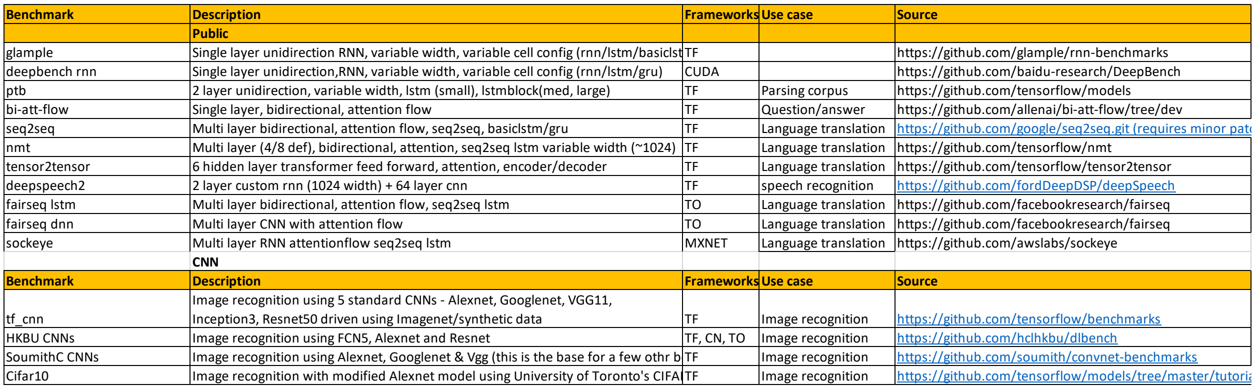
References

In no real order

Books:
Deep Learning (Goodfellow, Bengio and Courville)
Hands-on machine learning with Scitkit-Learn and Tensorflow (Aurelien Geron)
Web sites, papers and blogs:
http://www.wildml.com/
http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/
http://www.wildml.com/2016/08/rnns-in-tensorflow-a-practical-guide-and-undocumented-features/?utm_campaign=Revue%20newsletter&utm_medium=Newsletter&utm_source=The%20Wild%20Week%20in%20AI
http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/
http://karpathy.github.io/2015/05/21/rnn-effectiveness/
http://karpathy.github.io/neuralnets/
https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation.ipynb
http://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html
https://r2rt.com/recurrent-neural-networks-in-tensorflow-ii.html

https://r2rt.com/implementing-batch-normalization-in-tensorflow.html
https://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html
https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning/
https://blog.heuritech.com/2016/01/20/attention-mechanism/
https://richliao.github.io/supervised/classification/2016/12/26/textclassifier-RNN/
https://github.com/spitis
https://r2rt.com/
https://theneuralperspective.com/
http://imatge-upc.github.io/telecombcn-2016-dlcv/slides/D2L1-memory.pdf
http://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf
https://www.slideshare.net/BhaskarMitra3/a-simple-introduction-to-word-embeddings
http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/
http://colah.github.io/posts/2015-08-Backprop/
http://adventuresinmachinelearning.com/stochastic-gradient-descent/
https://distill.pub/2016/augmented-rnns/
http://neuralnetworksanddeeplearning.com/chap2.html
https://sudeepraja.github.io/Neural/
http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/
https://www.youtube.com/watch?v=UXW6Cs82UKo
http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf
https://arxiv.org/pdf/1409.3215.pdf
https://arxiv.org/pdf/1609.08144.pdf
https://arxiv.org/pdf/1508.04025.pdf
https://arxiv.org/pdf/1508.04395.pdf
http://colinraffel.com/publications/iclr2016feed.pdf
http://ai.stanford.edu/~wzou/emnlp2013_ZouSocherCerManning.pdf
https://arxiv.org/pdf/1412.6448.pdf
https://arxiv.org/abs/1301.3781
https://www.quora.com/How-does-word2vec-work-Can-someone-walk-through-a-specific-example
mikolov thesis: https://www.semanticscholar.org/paper/Language-Models-for-Automatic-Speech-Recognition-o-Mikolov-%C4%8Cernock%C3%BD/1975c4cd5ede97dc425eaa65d3d112468455bf1b
https://nlp.stanford.edu/pubs/glove.pdf
https://nlp.stanford.edu/projects/glove/
https://arxiv.org/pdf/1508.07909.pdf
https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b
http://cs231n.stanford.edu/

Useful public benchmark codes



Benchmark	Description	Frameworks	Use case	Source
	Public			
glample	Single layer unidirection RNN, variable width, variable cell config (rnn/lstm/basiclst	TF		https://github.com/glample/rnn-benchmarks
deepbench rnn	Single layer unidirection,RNN, variable width, variable cell config (rnn/lstm/gru)	CUDA		https://github.com/baidu-research/DeepBench
ptb	2 layer unidirection, variable width, lstm (small), lstmblock(med, large)	TF	Parsing corpus	https://github.com/tensorflow/models
bi-att-flow	Single layer, bidirectional, attention flow	TF	Question/answer	https://github.com/allenai/bi-att-flow/tree/dev
seq2seq	Multi layer bidirectional, attention flow, seq2seq, basiclstm/gru	TF	Language translation	https://github.com/google/seq2seq.git (requires minor pat
nmt	Multi layer (4/8 def), bidirectional, attention, seq2seq lstm variable width (~1024)	TF	Language translation	https://github.com/tensorflow/nmt
tensor2tensor	6 hidden layer transformer feed forward, attention, encoder/decoder	TF	Language translation	https://github.com/tensorflow/tensor2tensor
deepspeech2	2 layer custom rnn (1024 width) + 64 layer cnn	TF	speech recognition	https://github.com/fordDeepDSP/deepSpeech
fairseq lstm	Multi layer bidirectional, attention flow, seq2seq lstm	TO	Language translation	https://github.com/facebookresearch/fairseq
fairseq dnn	Multi layer CNN with attention flow	TO	Language translation	https://github.com/facebookresearch/fairseq
sockeye	Multi layer RNN attentionflow seq2seq lstm	MXNET	Language translation	https://github.com/awslabs/sockeye
	CNN			
Benchmark	**Description**	**Frameworks**	**Use case**	**Source**
tf_cnn	Image recognition using 5 standard CNNs - Alexnet, Googlenet, VGG11, Inception3, Resnet50 driven using Imagenet/synthetic data	TF	Image recognition	https://github.com/tensorflow/benchmarks
HKBU CNNs	Image recognition using FCN5, Alexnet and Resnet	TF, CN, TO	Image recognition	https://github.com/hclhkbu/dlbench
SoumithC CNNs	Image recognition using Alexnet, Googlenet & Vgg (this is the base for a few othr b	TF	Image recognition	https://github.com/soumith/convnet-benchmarks
Cifar10	Image recognition with modified Alexnet model using University of Toronto's CIFA	TF	Image recognition	https://github.com/tensorflow/models/tree/master/tutori

And
https://github.com/jonsafari/nmt-list
https://youtu.be/T8tQZChniMk