CS 337, Fall 2023
Batch Normalization, RNNs, LSTMs

Scribes: Akshat Goyal, Gargi Bakshi, Sanapati Hasini
P.Hemanth Naidu, R.Santhosh Reddy, M.Rishi Naik[*]
Edited by: Parshant Arora

October 3, 2023

1 Batch Normalization

1.1 Motivation

Recall, from linear regression, that preprocessing the inputs to have zero mean and unit variance (normalization) was the typical first step in the pipeline. *Why?*

- It helps constrain the function complexity of the hypotheses in the hypothesis plane.
- Standardization works well with gradient descent style optimizers by putting all parameters apriori in a similar space.

Internal Covariate Shift

In deep neural networks, variables in intermediate layers take values of varying magnitudes. When the parameters (variables) of a layer change, so do the distributions of inputs to the subsequent layers.

Internal covariate shift is defined as the change in the distribution of network activations due to the change in network parameters during training. One approach to address this issue is to use adaptive solvers like Adagrad, while an alternative method involves employing adaptive normalization techniques such as batch normalization.

Sergey Ioffe and Christian Szegedy proposed **Batch Normalization** to mitigate the problem of internal covariate shift, which can cause convergence challenges in deep neural networks, by re-centering and re-scaling the inputs of each layer.

[*]All scribes get equal credit for the final notes.

Note: It is argued that batch normalization does not reduce internal covariate shift, but rather makes the overall training dynamics smoother, which improves the performance.

1.2 Procedure

Batch normalization has two steps:

1. In each training iteration, from each layer's activations, subtract the mean and divide by the standard deviation.

2. After normalizing, apply a scaling coefficient and shift the inputs by an offset (both are learnable parameters).

The mean and standard deviation for a batch are computed using mini-batch statistics during training.
Let $\mathcal{B}$ denote a batch and let $\mathbf{x} \in \mathcal{B}$. $\mathrm{BN}(\mathbf{x})$ can be written as

$$\boxed{\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu_{\mathcal{B}}}}{\hat{\sigma_{\mathcal{B}}}} + \beta}$$

Batch mean:

$$\hat{\mu_{\mathcal{B}}} = \frac{\sum\limits_{\mathbf{x}\in\mathcal{B}} \mathbf{x}}{|\mathcal{B}|}$$

Batch variance:

$$\hat{\sigma_{\mathcal{B}}}^2 = \frac{\sum\limits_{\mathbf{x}\in\mathcal{B}} (\mathbf{x} - \hat{\mu_{\mathcal{B}}})^2}{|\mathcal{B}|} + \epsilon$$

Here, γ is the scaling coefficient, and β is the offset. These parameters are learned in the optimization process. ϵ (an arbitrarily small constant) is added for numerical stability.

At test time, we use the running averages of the mean and standard deviation values obtained during training.

Batch normalization is typically applied after the affine layer (fully connected layer) but before the non-linear activation.

1.3 Benefits of Batch Normalization

- Improves gradient flow through the network, makes more activation functions viable.
- Allows us to use much higher learning rates and be less careful about initialization.
- It also acts as a regularizer, in some cases eliminating the need for Dropout.

Batch normalization makes overall training dynamics smoother, doesn't necessarily reduce internal covariate shift. Batch normalization have worked best for CONV and regular deep Neural Networks. Other popular regularization techniques include the use of **Layer Normalization, Group Normalization**; **Residual Connections** help enable stable learning in very deep architectures.

1.4 Other Normalizations

- **Layer normalization**: Layer normalization normalizes the activations of a layer across all units in the layer. Layer normalization doesn't depend on batches and the normalization is per data point, so the exact same calculations are done during training time and test time.

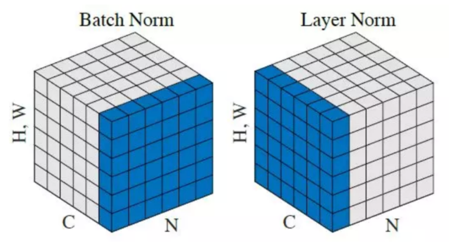


Figure 1: Layer Normalization

- **Group normalization**: Group normalization is a more generalized version of layer normalization, it divides the features into groups and normalizes the features within each group independently. The number of groups is a hyperparameter specified prior to training.

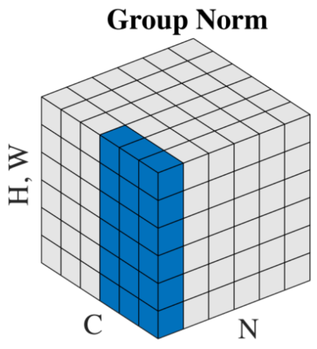


Figure 2: Group Normalization

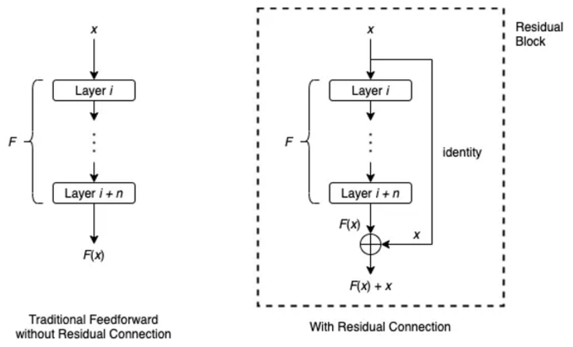


Figure 3: Residual Block

- **Residual connections**: Residual connections provide another path for data to reach the latter parts of the neural network by skipping some layers. It addresses the vanishing gradient problem and has led to the development of extremely deep networks, such as ResNets, that have achieved state-of-the-art results in many tasks.

2 Recurrent Neural Networks

So far, in feed-forward neural networks and CNNs, we have dealt with fixed-size inputs and fixed-size outputs. However, we often need to deal with inputs and outputs of varying lengths. RNNs come to our rescue here.

RNNs are neural networks that deal with sequential data and also give a variable-sized output. RNNs are distinguished by their memory, as they (unlike classical NNs and CNNs) take information from prior inputs and outputs to influence the current output. This memory is maintained through a hidden state.

2.1 RNN Use Cases

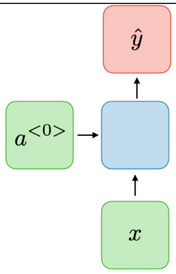
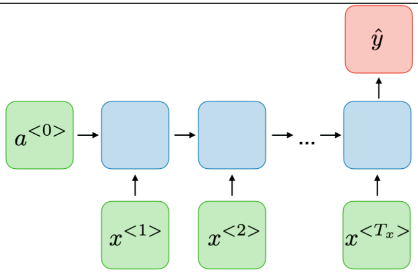
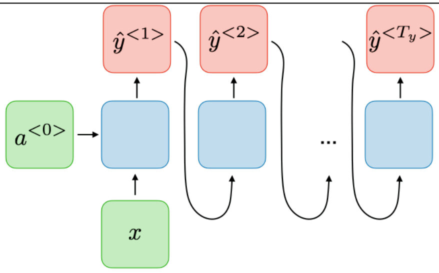
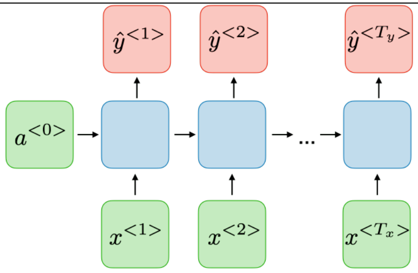
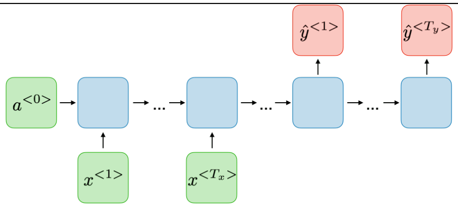
Type of input and output	Example task	RNN type	RNN Representation
Fixed-length input and output	Any NN or CNN task	one to one	
Variable length input and fixed-length output	Sentiment Analysis, Hate Speech Detection	many to one	 $\hat{y}$ $a^{<0>}$ ... $x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Fixed length input and variable length output	Image Captioning	one to many	 $\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... x
Variable length input and output, same size	Sequence Labelling, Part of speech tagging	many to many	 $\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... $x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Variable length input and output, different size	Translation	many to many	 $\hat{y}^{<1>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... ... $x^{<1>}$ $x^{<T_x>}$

Table 1: Common RNN use cases

2.2 Language Modelling

The language modeling problem focuses on predicting the next word in a sentence; given a word history $w_1, w_2, \ldots, w_{t-1}$, we want to find w^*

$$w^* = \arg\max_w P(w \,|\, w_1, w_2, \ldots, w_{t-1})$$

.

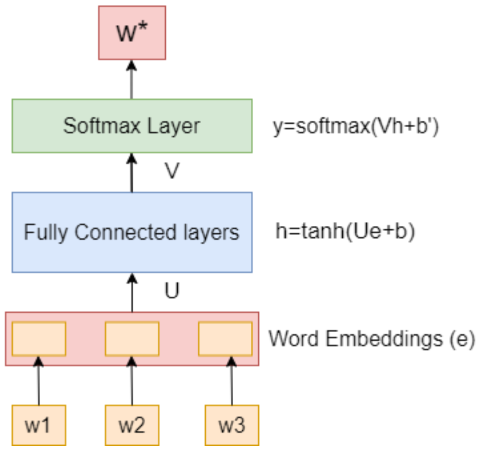


Figure 4: Language modeling using feed-forward neural networks

We cannot directly work with words so we use word embeddings. An embedding of a word is a representation of the word in a d-dimensional vector space. If we were to use basic feed-forward neural networks for the language modeling problem, we could only look at some n length history because feed-forward neural networks have fixed size inputs, so we lose information about the previous words. Such a model makes an n^{th} order Markovian assumption, that given the previous $n-1$ words, the probability of the n^{th} word is independent of the words prior to those $n-1$ words. So after predicting one word, we slide our window of inputs to predict the next.

However, RNNs that can work with variable-length inputs are more suited for this task.

2.3 RNN Architecture

RNNs maintain a hidden state, which remembers the information about past inputs. The output is predicted using the hidden state for the current time stamp. For each input, it uses the same parameters to predict the output. This reduces the complexity of the model, unlike other neural networks.

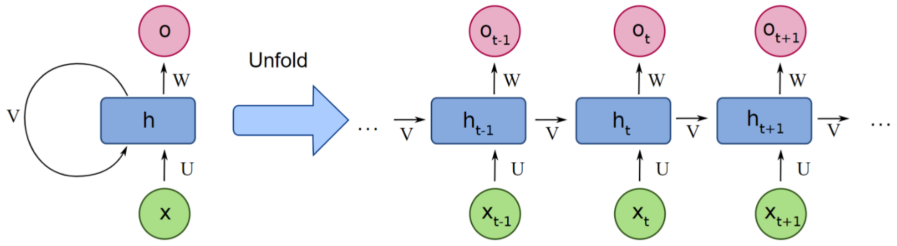


Figure 5: RNN Architecture

Consider a sample RNN for a classification task

$$\mathbf{h}_t = tanh(\mathbf{U}\mathbf{x}_t + \mathbf{V}\mathbf{h}_{t-1} + \mathbf{b})$$

$$\mathbf{o}_t = \mathbf{W}\mathbf{h}_t + b'$$

$$\hat{\mathbf{y}}_t = softmax(\mathbf{o}_t)$$

Here $\mathbf{U}, \mathbf{V}, \mathbf{W}, \mathbf{b}, b'$ are the parameters of the model which are shared across time steps.
$\mathbf{h}_t$ is the hidden state at the t^{th} timestep
$\mathbf{x}_t, \mathbf{o}_t$ are the input and output at the t^{th} timestep
$\mathbf{y}_t$ is the prediction probability distribution at the t^{th} timestep

2.4 Vanishing/Exploding Gradients

RNNs by design model temporal dependency, like a very deep neural network. Because of the large depth, the gradient, which is multiplicative across layers can vary greatly in magnitude causing the vanishing gradient or the exploding gradient problem.

The **exploding gradient** problem occurs when the gradients of loss function become very large during backpropagation, it shows that the model is unstable and unable to learn.

The **vanishing gradient** problem occurs when the gradients of loss function become very small as they are backpropagated, it hinders the model's capabilities to learn long-term dependencies.

2.5 Gradient Clipping

Gradient clipping is a technique used to deal with the exploding gradient problem. When performing back propagation we cap the maximum values(norm) for the gradient.

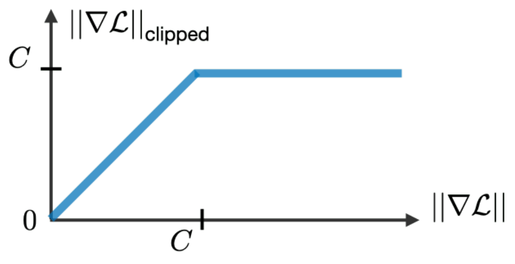


Figure 6: Gradient Clipping

3 Long-Short Term Memory Networks

Long-Short Term Memory Networks (**LSTM**s) is a variant of RNN, used in deep architectures specifically used to address the Vanishing-gradient problem. Unlike RNN, rather than applying an element wise non linearity to the affine transformation of inputs and recurrent units, LSTM consists of **gates** that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.These gates enable LSTM's to both **accumulate** and **forget** states conditioned on the context.

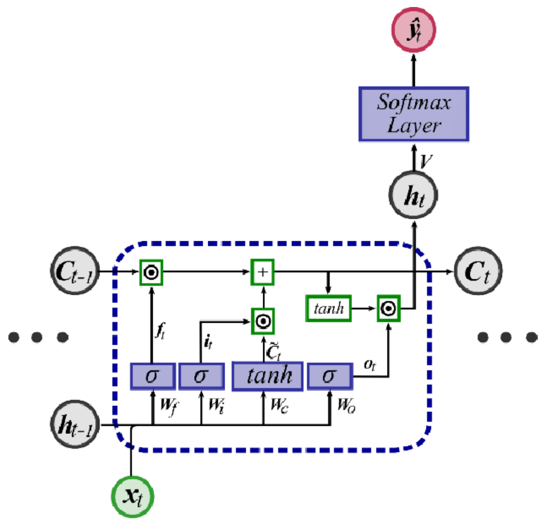
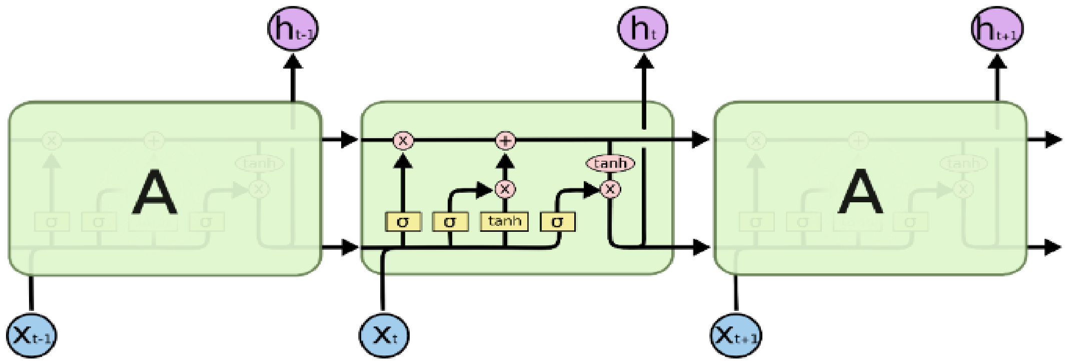


Figure 7: LSTM cell unit

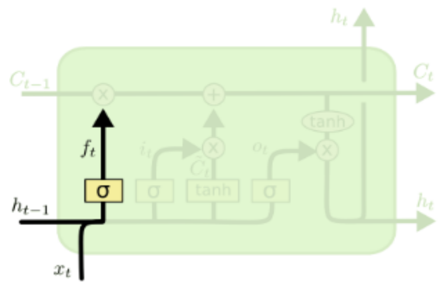


There are two states in LSTM, cell-state C_t and hidden-state h_t. Cell state is a memory of the LSTM cell and hidden state (cell output) is an output of this cell. The LSTM introduces three types of gates—**input gate, output gate, and forget gate**, values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

3.1 LSTM implementation

3.1.1 Forget gate

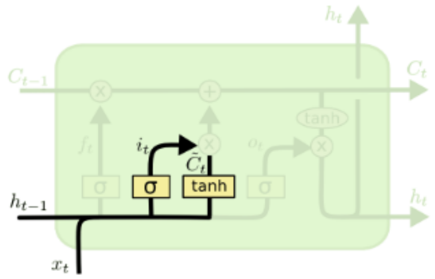
This gate decides what information should be thrown away or kept. The information from the previous hidden state and information from the current input is passed through the sigmoid function. It looks at h_{t-1} and current input x_t and outputs a number between 0 and 1 for each number in the cell state C_{t-1}, indicating whether to retain 1 or discard 0 the corresponding information from the cell state.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

3.1.2 Input gate

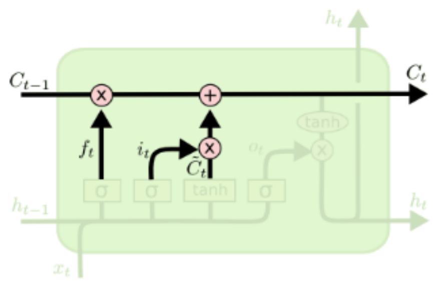
The input gate decides what relevant information can be added from the current step. Sigmoid layer decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

3.1.3 Cell State

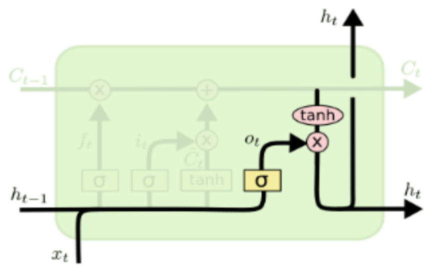
It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

3.1.4 Output gate

The output gate determines what the next hidden state should be using the output and the updated cell-state. Sigmoid layer decides what parts of the cell state we're going to output. Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate.

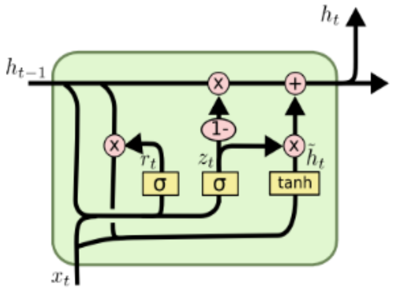


$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

where operator * denotes the Hadamard product, f_t is the forget gate, i_t is the input gate, o_t is the output gate, and C_t and h_t are the updated cell states. The non-linearities used are sigmoid and tanh.

4 Gated Recurrent Unit

A slight variation of the LSTM is the **Gated Recurrent Unit**(GRU), it combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models.



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