Advanced Deep Learning

(IISc Bengaluru, 2020)

Himanshu Singh

Github

Notations

$\boldsymbol{x} \in \Re^D$	Input data.
$\boldsymbol{y} \in \Im^C$	Neural Network target.
$\hat{\boldsymbol{y}} \in \Re^C$	Model Outputs.
$\boldsymbol{e}, \boldsymbol{h} \in \Re^d$	Hidden model representation/embeddings.
Φ	Collections of Learnable parameters.
$E(\boldsymbol{y}, \hat{\boldsymbol{y}})$	Error function used in Model Training.
$\{(x_1, y_1), \ldots, (x_N, y_N)\}$	Labelled Training data.
$q = \{1, 2, \ldots, Q\}$	Iteration Index.
$t = \{1, 2, \ldots, T\}$	Discrete Time Index.
$l = \{1, 2, \ldots, L\}$	Layer Index.
η	Learning rate (hyper parameter).
N_b	Mini Batch.
B	No. Of Mini batches.

Learning in Feed-forward Networks

Stochastic Gradient Descent

Initialize model parameters

for $q = \{1, 2, \ldots, Q\}$

for $b = \{1, 2, \ldots, B\}$

$$\Phi^{q,b} = \Phi^{q,b-1} - \eta \left. \frac{\partial E(\{y_{b,1}, \ldots y_{b,N_b}\}, \{\hat{y_{b,1}}, \ldots, \hat{y_{b,N_b}}\})}{\partial \Phi} \right|_{\Phi = \Phi^{q,b-1}}$$

$$\Phi^{q+1,0} = \Phi^{q,B}$$

return $\Phi^{*} = \Phi^{Q,B}$

Gradients Algorithms

- **Momentum**

$$v_t = \gamma v_{t-1} + \eta \nabla_{\Phi} E(\Phi)$$
$$\Phi_t = \Phi_{t-1} - v_t$$

- **Nestrov Accelarated Gradient**

$$v_t = \gamma v_{t-1} + \eta \nabla_{\Phi} E(\Phi - \gamma v_{t-1})$$
$$\Phi_t = \Phi_{t-1} - v_t$$

- **Adagrad**

$$\Phi_{t+1} = \Phi_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$
$$g_t = \nabla \Phi E(\Phi)$$

G_t is sum of square of gradients upto time t

- **Adadelta/ RMSprop**

$$\Phi_{t+1} = \Phi_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t$$
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g^2(t)$$

- **Adam**

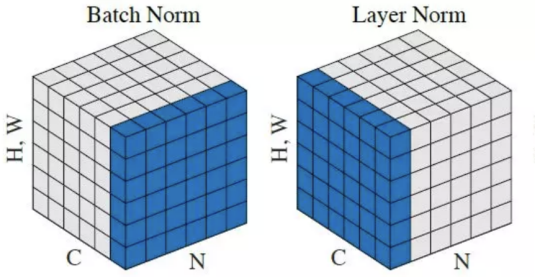
$$\Phi_{t+1} = \Phi_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad \hat{m}_t = \frac{m_t}{1 = \beta_1^t}$$

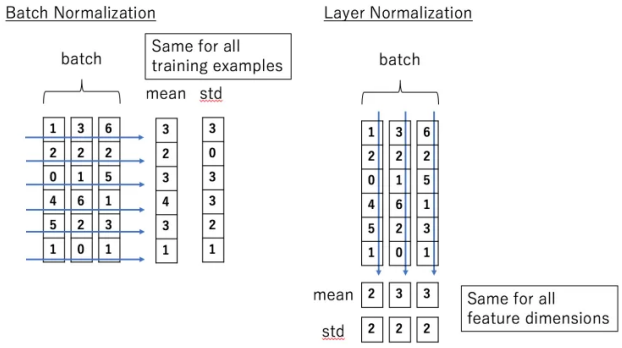
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \qquad \hat{v}_t = \frac{v_t}{1 = \beta_2^t}$$

Normalization

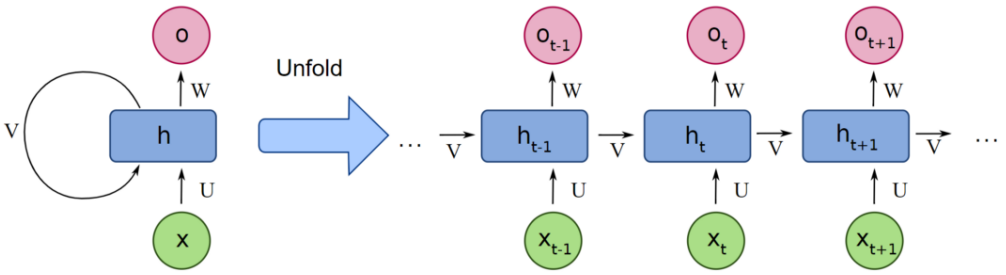
Batch Normalization calculates the mean and variance across all instances in the batch. It is widely used in Convolutional Neural Networks (CNNs) as it can accelerate training and improve generalization. In the formula for batch normalization, given a batch of activations for a specific layer, it first calculates the mean and standard deviation for the batch. Then, it subtracts the mean and divides by the standard deviation to normalize the values. Following normalization, batch normalization applies a scale factor "gamma" and shift factor "beta". These two parameters are learnable and allow the layer to undo the normalization if it finds it's not useful. Batch Normalization reduces the “internal covariate shift”, it also influences the smoothness of the optimization landscape, simplifying the training process. It removes the interdependency between layers during training



Layer Normalization normalizes the activations along the feature direction instead of mini-batch direction. It calculates the mean and variance for each instance separately, over all the features. Unlike batch normalization, it doesn't depend on the batch size, so it's often used in recurrent models where batch normalization performs poorly.
Layer normalization computes the mean and standard deviation across each individual observation instead (over all channels in case of images or all features in case of an MLP) rather than across the batch. This makes it batch-size independent and can therefore be used in models like RNNs or in transformer models.



Recurrent Models



$$\boldsymbol{a}^{1}(t)=\boldsymbol{U}\,\boldsymbol{x}(t)+\boldsymbol{V}\boldsymbol{h}(t-1)+\boldsymbol{b}^{1}$$
$$\boldsymbol{h}(t)=\tanh(\boldsymbol{a}^{1}(t))$$
$$\boldsymbol{a}^{2}(t)=\boldsymbol{W}\,\boldsymbol{h}(t)+\boldsymbol{b}^{2}$$
$$\hat{\boldsymbol{y}}(t)=sigmoid(\boldsymbol{a}^{2}(t))$$

Back-propagation through Time

Error function is computed at every time-instant

$$E=\sum_{t}E(\boldsymbol{y}(t),\hat{\boldsymbol{y}}(t))$$

Output Activation

$$\frac{\partial E}{\partial \boldsymbol{a}^{2}(t)}=\hat{\boldsymbol{y}}(t)-\boldsymbol{y}(t)\qquad for\ t=1,2,\ldots,T$$

Hidden Activation at last instant T

$$\frac{\partial E}{\partial \boldsymbol{h}(T)}=\boldsymbol{W}^{T}\frac{\partial E}{\partial \boldsymbol{a}^{2}(T)}$$

Hidden activations for previous instances t = T-1,..., 1

$$\frac{\partial E}{\partial \boldsymbol{h}(t)}=\boldsymbol{W}^{T}\frac{\partial E}{\partial \boldsymbol{a}^{2}(t)}+\frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)}\frac{\partial E}{\partial \boldsymbol{h}(t+1)}$$

$$=\underbrace{\boldsymbol{W}^{T}\frac{\partial E}{\partial \boldsymbol{a}^{2}(t)}}_{Error\ \text{from}\ output\ at\ current\ time\ t}+\underbrace{\boldsymbol{V}^{T}diag(1-(\boldsymbol{h}(t+1))^{.2})\frac{\partial E}{\partial \boldsymbol{h}(t+1)}}_{Error\ \text{from}\ Recurrence}$$

The derivative wrt output layer weights

$$\frac{\partial E}{\partial \boldsymbol{W}}=\sum_{t}\frac{\partial E}{\partial \boldsymbol{a}^{2}(t)}\boldsymbol{h}(t)^{T}$$

$$\frac{\partial E}{\partial \boldsymbol{b}^{2}}=\sum_{t}\frac{\partial E}{\partial \boldsymbol{a}^{2}(t)}$$

Transferring derivatives to input/hidden layer

$$\frac{\partial E}{\partial \boldsymbol{a}^{1}(t)}=diag(1-\boldsymbol{h}(t)^{.2})\frac{\partial E}{\partial \boldsymbol{h}(t)}$$

Derivative wrt input/hidden layer weights

$$\frac{\partial E}{\partial \boldsymbol{U}}=\sum_{t}\frac{\partial E}{\partial \boldsymbol{a}^{1}(t)}\boldsymbol{x}(t)^{T}$$

$$\frac{\partial E}{\partial \boldsymbol{V}}=\sum_{t}\frac{\partial E}{\partial \boldsymbol{a}^{1}(t)}\boldsymbol{h}(t)^{T}$$

$$\frac{\partial E}{\partial \boldsymbol{b}^{1}}=\sum_{t}\frac{\partial E}{\partial \boldsymbol{a}^{1}(t)}$$

Issue: Long-Term dependency issues. Gradients tends to vanish or explode

Long Short Term Memory (LSTM)

Forget Gate

$$a_t^f = W^f x_t + U^f h_{t-1} + b^f$$
$$f_t = \sigma(a_t^f)$$

Input Gate

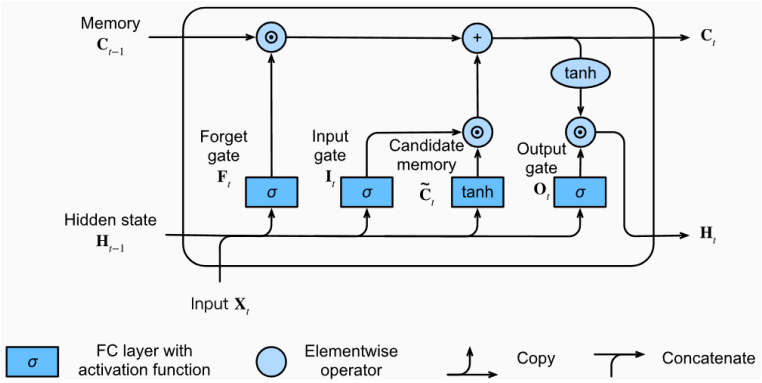
$$a_t^i = W^i x_t + U^i h_{t-1} + b^i$$
$$i_t = \sigma(a_t^i)$$

Cell state

$$a_t^c = W^c x_t + U^c h_{t-1} + b^c$$
$$\tilde{c}_t = \tanh(a_t^c)$$
$$c_t = (i_t \odot \tilde{c}_t) + (f_t \odot c_{t-1})$$

Output Gate

$$a_t^o = W^o x_t + U^o h_{t-1} + b^o$$
$$o_t = \sigma(a_t^c)$$
$$h_t = \tanh(c_t) \odot o_t$$



The cell state is meant to encode a kind of aggregation of data from all previous time-steps that have been processed, while the hidden state is meant to encode a kind of characterization of the previous time-step's data.

Encoder Decoder Models

$$h(t) = f(h(t-1), x(t))$$
$$e = f'(h(1), h(2), \ldots, h(T))$$

The encoder can have multiple layer of RNN/LSTM. For simplicity, $e = h(T)$

Encoder: Convert sequence $x = \{x(1), x(2), \ldots, x(T)\}$ to vector e

Decoder: Convert vector embedding from encoder to output sequence $\hat{y} = \{\hat{y}(1), \ldots, \hat{y}(S)\}$

$$p(\hat{y}) = \prod_{s=1}^{S} p(\hat{y}(s) | \hat{y}(1), \ldots, \hat{y}(s-1))$$

Decoder assumption

$$\begin{aligned} p(\hat{y}(s) | \hat{y}(1), \ldots, \hat{y}(s-1)) &= p(\hat{y}(s) | \hat{y}(s-1), e) \\ &= \text{softmax}(V \hat{y}(s-1) + Rc(s-1) + Te + d) \\ c(s) &= f(c(s-1), e) \end{aligned}$$

Modification to encoder-decoder model

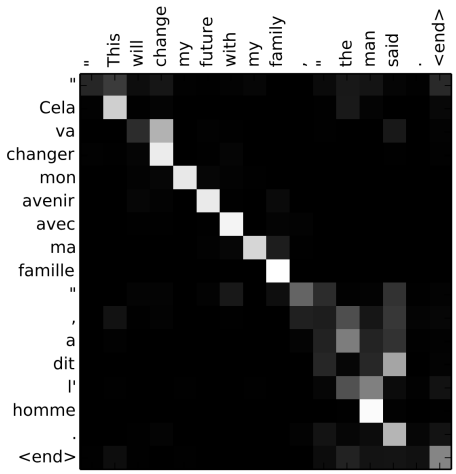
$$\begin{aligned} p(\hat{y}(s) | \hat{y}(1), \ldots, \hat{y}(s-1)) &= p(\hat{y}(s) | \hat{y}(s-1), e(s)) \\ &= \text{softmax}(V \hat{y}(s-1) + Rc(s-1) + Te(s) + d) \end{aligned}$$

where, $e(s) = \sum_{t=1}^{T} \alpha(s,t) h(t)$, $\alpha(s,t)$ captures the contribution of input at time t with output s.

Implementing this automatically using network-in network.

$$\hat{\alpha}(s,t) = A[c(s-1); h(t)]$$
$$\alpha(s,t) = \sigma(\hat{\alpha}(s,t)) = \frac{e^{\hat{\alpha}(s,t)}}{\sum_{t'} e^{\hat{\alpha}(s,t')}}$$

$\alpha(s,t)$ are called attention weights.



Multi-Head Attention

Having more than one attention heads.

$$\hat{\alpha}_1(s,t)=A_1[c(s-1);h(t)] \qquad e_1(s)=\sum_t \alpha_1(s,t)h(t)$$

...

$$\hat{\alpha}_H(s,t)=A_H[c(s-1);h(t)] \qquad e_H(s)=\sum_t \alpha_H(s,t)h(t)$$

$$e(s)=[e_1(s);\dots;e_H(s)]$$

Transformers

<u>Encoder:</u>

- Layer Norm
- Self Attention (Multi-Head)
- Pointwise feed-forward

$x(1),x(2),\dots,x(T)$ denote the input

$e^l(1),e^l(2),\dots,e^l(T)$ encoded outputs at layer l

$$\bar{E}^{l-1}=Layernorm([e^{l-1}(1),\dots,e^{l-1}(T)]^T)\in\Re^{T\times D}$$

$$Layernorm(e^l(t))=\frac{\alpha^l}{\sigma_{e^l(t)}}\odot(e^l(t)-\mu_{e^l(t)})+\beta^l$$

Query, Key and Value

$$Q_h^l=\bar{E}^{l-1}W_h^{l,Q}+\mathbf{1}(b_h^{l,Q})^T \qquad \in\Re^{T\times d}$$
$$K_h^l=\bar{E}^{l-1}W_h^{l,K}+\mathbf{1}(b_h^{l,K})^T \qquad \in\Re^{T\times d}$$
$$V_h^l=\bar{E}^{l-1}W_h^{l,V}+\mathbf{1}(b_h^{l,V})^T \qquad \in\Re^{T\times d}$$

$W_h^{l,Q},W_h^{l,K},W_h^{l,V}\in\Re^{D\times d}$ $\quad b_h^{l,Q},b_h^{l,Q},b_h^{l,Q}\in\Re^{d\times 1}$

$h=\{1,2,\dots,H\}$ $\quad d=\frac{D}{H}$ $\quad \mathbf{1}\in\Re^{T\times 1}$ all ones

Multi-Head Self Attention

$$\hat{A}_h^l=Q_h^l(K_h^l)^T \qquad \in\Re^{T\times T}$$

$$A_h^l=\text{softmax}\left(\frac{\hat{A}_h^l}{\sqrt{d}}\right)$$

$$C_h^l=A_h^lV_h^l \qquad \in\Re^{T\times d}$$

Context Vector from self-Attention

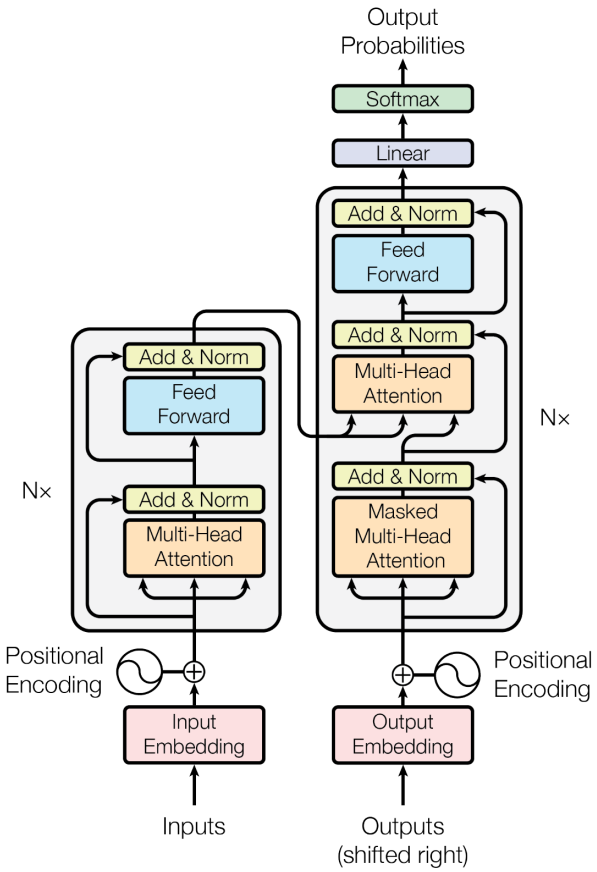
$$C^l=[C_1^l;\dots;C_H^l] \qquad \in\Re^{T\times D}$$

Position wise feed-forward layer

$$E_{ff}^l=ReLU(C^lW_{ff}^l+\mathbf{1}b_{ff}^T) \qquad \in\Re^{T\times d_{ff}}$$

Encoder layer output

$$[e^l(1),\dots,e^l(T)]=E_{ff}^lW_{of}^l+\mathbf{1}(b_{of}^l)^T \qquad \in\Re^{T\times D}$$



<u>Positional Encoding</u>

Even if we jumble the words, the attention tries to figure out relation between the words of the document, there is position awareness. Since different sequences have different length, we want to encode them such that length of sequence and order of sequence is communicated to model.

$$p(t) \in \Re^D$$

$$p_i(t) = \begin{cases} \sin(\omega_k t), & if\ i = 2k \\ \cos(\omega_k t), & if\ i = 2k+1 \end{cases} \qquad k \in \left\{1, 2, \ldots \frac{D}{2}\right\}$$

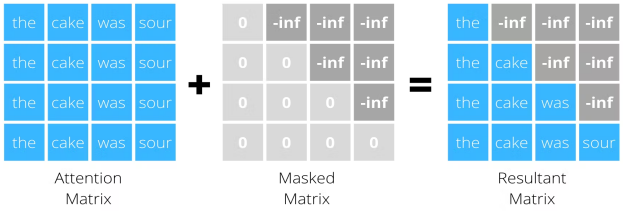
$$\omega_k = \frac{1}{10000^{2k/D}}$$

$$x(t) = x(t) + p(t)$$

Decoder

Masked Self Attention: Mask make output dependencies casual, only the past is used to encode the attention.

$$\text{Softmax}\left\{\frac{QK^T}{\sqrt{d}}\right\}V \rightarrow \text{Softmax}\left\{\text{Mask} + \frac{QK^T}{\sqrt{d}}\right\}V$$



Encoder Decoder Attention: Use the key and value matrices from the last layer of encoder.

$$Q_h^p = \bar{D}^{p-1} W_h^{p,Q} + \mathbf{1}\left(b_h^{p,Q}\right)^T \qquad \in \Re^{S \times d}$$

$$K_h^p = E^L W_h^{p,K} + \mathbf{1}\left(b_h^{p,K}\right)^T \qquad \in \Re^{T \times d}$$

$$V_h^p = E^L W_h^{p,V} + \mathbf{1}\left(b_h^{p,V}\right)^T \qquad \in \Re^{T \times d}$$

$$D_h^p = \text{softmax}\left(\frac{Q_h^p \left(K_h^p\right)^T}{\sqrt{d}}\right) V_h^p \qquad \in \Re^{S \times d}$$

$$W_h^{p,Q}, W_h^{p,K}, W_h^{p,V} \in \Re^{D \times d} \qquad b_h^{p,Q}, b_h^{p,Q}, b_h^{p,Q} \in \Re^{d \times 1}$$

$$h = \{1, 2, \ldots, H\} \qquad d = \frac{D}{H} \qquad \mathbf{1} \in \Re^{T \times 1} \text{ all ones}$$

Representation learning/ Data Visualization

Learning a lower dimensional representation which may generate transform applicable to newer data or may be dataset specific dimensionality reduction.

Neighbourhood preserving dimensionality reduction $x_1, x_2, \dots, x_N \in \Re^D \rightarrow y_1, y_2, \dots, y_N \in \Re^d$.

Neighbourhood is based on some distance metric. If two sample are close in original space, this has to be preserved in lower dimensionality as well.

$$d_{i,j} = \frac{\|x_i - x_j\|^2}{2\sigma^2}$$

Stochastic Neighbourhood Embedding (SNE)

$$p_{j|i} = \frac{\exp(-d_{ij}^2)}{\sum_{k \neq i} \exp(-d_{ik}^2)} \qquad p_{i|i} = 0$$

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} \exp(-\|y_i - y_k\|^2)}$$

$p_{j|i}$ is probability distribution based on neighbourhood, $q_{j|i}$ is distribution on the lower space. In lower dimension we choose same Gaussian style distribution but with 0.5 variance.

Error function

$$E = \sum_j \sum_{i \neq j} KL(p_{j|i} \| q_{j|i}) = \sum_j \sum_{i \neq j} p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

$$\frac{\partial E}{\partial y_i} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

Can be solved using Gradient descent.

Having a uniform variance may ne harmful. We can use data specific variance. $d_{i,j} = \frac{\|x_i - x_j\|^2}{2\sigma_i^2}$

Variance can be chosen to have uniform entropy $H_i = -\sum_j p_{j|i} \log p_{j|i}$

Crowding problem: Data in high dimensions tend to loose their spread and therefore tend to crowd in lower dimension

t-SNE

Student t-distribution with 1 degree of freedom

$$q_{j|i} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq i} (1 + \|y_i - y_k\|^2)^{-1}}$$

$$\frac{\partial E}{\partial y_i} = 2 \sum_j (p_{j|i} - q_{j|i})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

Hyper parameter: Perplexity, learning rate, number of iteration

Pros: Data neighbourhood preserving and relatively intuitive in visualizing data.

Cons: Iterative learning, does not provide a transform on unseen data.

Lecture 11