Lecture 26: Final Exam Review, Part 1

Mark Hasegawa-Johnson

University of Illinois

ECE 417: Multimedia Signal Processing

Outline

Final Exam: General Structure

- About twice as long as a midterm (i.e., 8-10 problems with 1-3 parts each)
- You'll have 3 hours for the exam (December 13, 8-11am)
- The usual rules: no calculators or computers, two sheets of handwritten notes, you will have two pages of formulas provided on the exam, published by the Friday before the exam.

Final Exam: Topics Covered

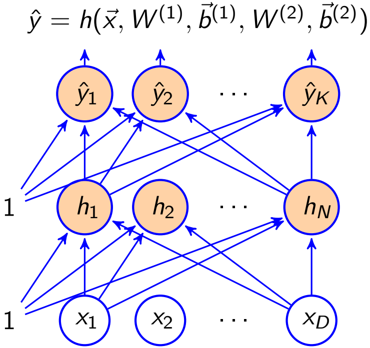
- 17%: Material from exam 1 (signal processing)
- 17%: Material from exam 2 (probability)
- 66%: Material from the last third of the course (neural networks)

Material from the last third of the course

- Neural networks & back-propagation
- CNN & Faster RCNN
- Partial derivatives & RNN
- LSTM

Outline

Two-Layer Feedforward Neural Network



$$\hat{y}_k = g(\xi_k^{(2)})$$

$$\xi_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = g(\xi_k^{(1)})$$

$$\xi_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

How to train a neural network

1. Find a **training dataset** that contains n examples showing the desired output, $\vec{y}_i$, that the NN should compute in response to input vector $\vec{x}_i$:

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

2. Randomly **initialize** $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
3. Perform **forward propagation**: find out what the neural net computes as $\hat{y}_i$ for each $\vec{x}_i$.
4. Define a **loss function** that measures how badly $\hat{y}$ differs from $\vec{y}$.
5. Perform **back propagation** to find the derivative of the loss w.r.t. $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
6. Perform **gradient descent** to improve $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
7. Repeat steps 3-6 until convergence.

Gradient Descent = Local Optimization

Given an initial W, b, find new values of W, b with lower error.

$$w_{kj}^{(1)} \leftarrow w_{kj}^{(1)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(1)}}$$

$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(2)}}$$

η =Learning Rate

- If η too large, gradient descent won't converge. If too small, convergence is slow.
- Second-order methods like Newton's method, L-BFGS and Adam choose an optimal η at each step, so they're MUCH faster.

Loss Function: How should y be "similar to" $\hat{y}$?

Minimum Mean Squared Error (MMSE)

$$W^*, b^* = \arg\min \mathcal{L} = \arg\min \frac{1}{2n}\sum_{i=1}^{n} \|\vec{y}_i - \hat{y}(\vec{x}_i)\|^2$$

MMSE Solution: $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$

If the training samples $(\vec{x}_i, \vec{y}_i)$ are i.i.d., then

$$\lim_{n\rightarrow\infty} \mathcal{L} = \frac{1}{2} E\left[\|\vec{y} - \hat{y}\|^2\right]$$

which is minimized by

$$\hat{y}_{MMSE}(\vec{x}) = E\left[\vec{y}|\vec{x}\right]$$

Binary Cross Entropy

Suppose we treat the neural net output as a noisy estimator, $\hat{p}_{Y|\vec{X}}(y|\vec{x})$, of the unknown true pmf $p_{Y|\vec{X}}(y|\vec{x})$:

$$\hat{y}_i = \hat{p}_{Y|\vec{X}}(1|\vec{x}),$$

so that

$$\hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) = \begin{cases} \hat{y}_i & y_i = 1 \\ 1 - \hat{y}_i & y_i = 0 \end{cases}$$

The binary cross-entropy loss is the negative log probability of the training data, assuming i.i.d. training examples:

$$\begin{aligned} \mathcal{L}_{BCE} &= -\frac{1}{n}\sum_{i=1}^{n} \ln \hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) \\ &= -\frac{1}{n}\sum_{i=1}^{n} y_i \left(\ln \hat{y}_i\right) + (1 - y_i)\left(\ln(1 - \hat{y}_i)\right) \end{aligned}$$

The Derivative of BCE

BCE is useful because it has the same solution as MSE, without allowing the sigmoid to suffer from vanishing gradients. Suppose $\hat{y}_i = \sigma(\xi_i)$.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \xi_i} &= -\frac{1}{n}\left(y_i \frac{\partial \ln \sigma(\xi_i)}{\partial \xi_i} + (1-y_i)\frac{\partial \ln(1-\sigma(\xi_i))}{\partial \xi_i}\right) \\
&= -\frac{1}{n}\left(y_i \frac{\dot{\sigma}(\xi_i)}{\sigma(\xi_i)} - (1-y_i)\frac{1-\dot{\sigma}(\xi_i)}{1-\sigma(\xi_i)}\right) \\
&= -\frac{1}{n}\left(y_i \frac{\hat{y}_i(1-\hat{y}_i)}{\hat{y}_i} - (1-y_i)\frac{\hat{y}_i(1-\hat{y}_i)}{1-\hat{y}_i}\right) \\
&= -\frac{1}{n}(y_i - \hat{y}_i)
\end{aligned}$$

where the last line is true because $y_i \in \{0, 1\}$.

Multinomial Classifier

Suppose, instead of just a 2-class classifier, we want the neural network to classify $\vec{x}$ as being one of K different classes. There are many ways to encode this, but one of the best is

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{bmatrix}, \quad y_k = \begin{cases} 1 & k = k^* \quad (k \text{ is the correct class}) \\ 0 & \text{otherwise} \end{cases}$$

A vector $\vec{y}$ like this is called a "one-hot vector," because it is a binary vector in which only one of the elements is nonzero ("hot"). This is useful because minimizing the MSE loss gives:

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \begin{bmatrix} \hat{p}_{Y_1|\vec{X}}(1|\vec{x}) \\ \hat{p}_{Y_2|\vec{X}}(1|\vec{x}) \\ \vdots \\ \hat{p}_{Y_K|\vec{X}}(1|\vec{x}) \end{bmatrix},$$

One-hot vectors and Cross-entropy loss

The cross-entropy loss, for a training database coded with one-hot vectors, is

$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ki}\ln \hat{y}_{ki}$$

This is useful because:

1. **Like MSE, Cross-Entropy has an asymptotic global optimum at:** $\hat{y}_k \rightarrow p_{Y_k|\vec{X}}(1|\vec{x})$.
2. **Unlike MSE, Cross-Entropy with a softmax nonlinearity suffers no vanishing gradient problem.**

Softmax Nonlinearity

The multinomial cross-entropy loss is only well-defined if $0 < \hat{y}_{ki} < 1$, and it is only well-interpretable if $\sum_k \hat{y}_{ki} = 1$. We can guarantee these two properties by setting

$$\begin{aligned}\hat{y}_k &= \operatorname*{softmax}_k \left(W\vec{h}\right)\\ &= \frac{\exp(\bar{w}_k \vec{h})}{\sum_{\ell=1}^{K} \exp(\bar{w}_\ell \vec{h})},\end{aligned}$$

where $\bar{w}_k$ is the k^{th} row of the W matrix.

Sigmoid is a special case of Softmax!

$$\text{softmax}_k\left(W\vec{h}\right) = \frac{\exp(\bar{w}_k\vec{h})}{\sum_{\ell=1}^{K}\exp(\bar{w}_\ell\vec{h})}.$$

Notice that, in the 2-class case, the softmax is just exactly a logistic sigmoid function:

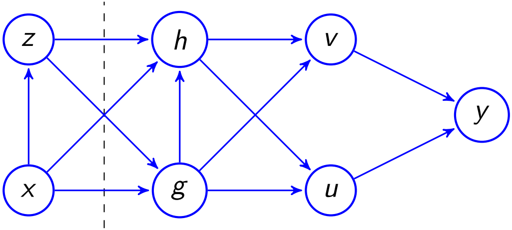
$$\text{softmax}_1(W\vec{h}) = \frac{e^{\bar{w}_1\vec{h}}}{e^{\bar{w}_1\vec{h}} + e^{\bar{w}_2\vec{h}}} = \frac{1}{1 + e^{-(\bar{w}_1 - \bar{w}_2)\vec{h}}} = \sigma\left((\bar{w}_1 - \bar{w}_2)\vec{h}\right)$$

so everything that you've already learned about the sigmoid applies equally well here.

The Total Derivative Rule

The **total derivative rule** says that the derivative of the output with respect to any one input can be computed as the sum of partial times total, summed across all paths from input to output:

$$\frac{\partial y(x,z)}{\partial x} = \left(\frac{dy}{dg}\right)\left(\frac{\partial g(x,z)}{\partial x}\right) + \left(\frac{dy}{dh}\right)\left(\frac{\partial h(g,x,z)}{\partial x}\right)$$



The Back-Propagation Algorithm

$$W^{(2)} \leftarrow W^{(2)} - \eta \nabla_{W^{(2)}} \mathcal{L}, \qquad W^{(1)} \leftarrow W^{(1)} - \eta \nabla_{W^{(1)}} \mathcal{L}$$

$$\nabla_{W^{(2)}} \mathcal{L} = \sum_{i=1}^{n} \nabla_{\vec{\xi}_i^{(2)}} \mathcal{L} \vec{h}_i^T, \qquad \nabla_{W^{(1)}} \mathcal{L} = \sum_{i=1}^{n} \nabla_{\vec{\xi}_i^{(1)}} \mathcal{L} \vec{x}_i^T$$

$$\nabla_{\vec{\xi}_i^{(2)}} \mathcal{L} = \frac{1}{n}(\hat{y}_i - \vec{y}_i), \qquad \nabla_{\vec{\xi}_i^{(1)}} \mathcal{L} = \dot{\sigma}(\vec{\xi}_i^{(1)}) \odot W^{(2),T} \nabla_{\vec{\xi}_i^{(2)}} \mathcal{L}$$

Derivative of a sigmoid

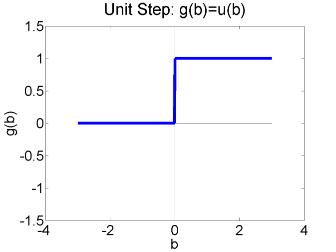
The derivative of a sigmoid is pretty easy to calculate:

$$h = \sigma(\xi) = \frac{1}{1+e^{-\xi}}, \quad \frac{dh}{d\xi} = \dot{\sigma}(\xi) = \frac{e^{-\xi}}{(1+e^{-\xi})^2}$$

An interesting fact that's extremely useful, in computing back-prop, is that if $h = \sigma(\xi)$, then we can write the derivative in terms of h, without any need to store ξ:

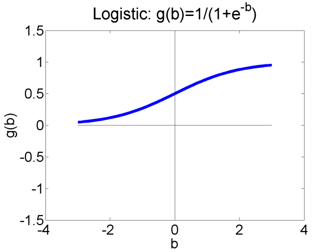
$$\begin{aligned}
\frac{d\sigma}{d\xi} &= \frac{e^{-\xi}}{(1+e^{-\xi})^2} \\
&= \left(\frac{1}{1+e^{-\xi}}\right)\left(\frac{e^{-\xi}}{1+e^{-\xi}}\right) \\
&= \left(\frac{1}{1+e^{-\xi}}\right)\left(1 - \frac{1}{1+e^{-\xi}}\right) \\
&= \sigma(\xi)(1-\sigma(\xi)) \\
&= h(1-h)
\end{aligned}$$

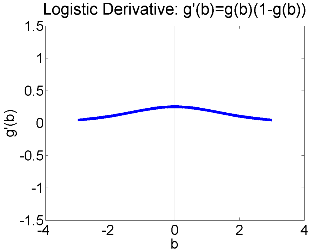
Step function and its derivative



- The derivative of the step function is the Dirac delta, which is not very useful in backprop.

Logistic function and its derivative





Signum and Tanh

The signum function is a signed binary nonlinearity. It is used if, for some reason, you want your output to be $h \in \{-1, 1\}$, instead of $h \in \{0, 1\}$:

$$\text{sign}(b) = \begin{cases} -1 & b < 0 \\ 1 & b > 0 \end{cases}$$

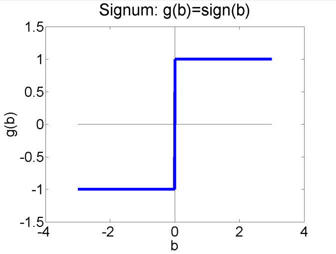
It is usually approximated by the hyperbolic tangent function (tanh), which is just a scaled shifted version of the sigmoid:

$$h = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}} = \frac{1 - e^{-2\xi}}{1 + e^{-2\xi}} = 2\sigma(2\xi) - 1$$

and which has a scaled version of the sigmoid derivative:

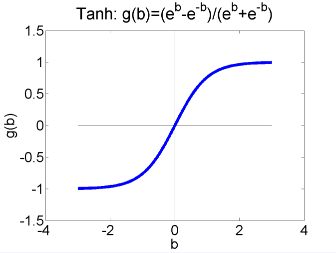
$$\frac{d \tanh(\xi)}{d\xi} = \left(1 - \tanh^2(\xi)\right)$$

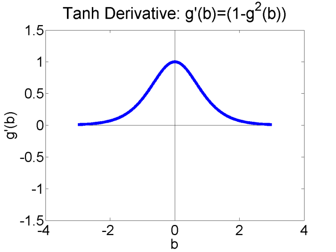
Signum function and its derivative



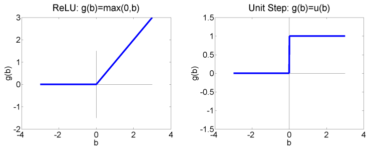
- The derivative of the signum function is the Dirac delta, which is not very useful in backprop.

Tanh function and its derivative





A solution to the vanishing gradient problem: ReLU



The most ubiquitous solution to the vanishing gradient problem is to use a ReLU (rectified linear unit) instead of a sigmoid. The ReLU is given by

$$\mathrm{ReLU}(\xi) = \begin{cases} b & \xi \geq 0 \\ 0 & \xi \leq 0, \end{cases}$$

and its derivative is

$$\frac{d\mathrm{ReLU}(\xi)}{d(\xi)} = u(\xi)$$

Outline

How to achieve shift invariance: Convolution

Instead of using vectors as layers, let's use images.

$$\xi^{(l)}[m,n,d] = \sum_{c}\sum_{m'}\sum_{n'} w^{(l)}[m',n',c,d]h^{(l-1)}[m-m',n-n',c]$$

where

- $\xi^{(l)}[m,n,c]$ and $h^{(l)}[m,n,c]$ are excitation and activation (respectively) of the $(m,n)^{\text{th}}$ pixel, in the c^{th} channel, in the l^{th} layer.
- $w^{(l)}[m,n,c,d]$ are weights connecting c^{th} input channel to d^{th} output channel, with a shift of m rows, n column.

Convolution forward, Correlation backward

In signal processing, we defined $x[n] * h[n]$ to mean $\sum h[m]x[n-m]$. Let's use the same symbol to refer to this multi-channel 2D convolution:

$$\begin{aligned}\xi^{(l)}[m,n,d] &= \sum_c \sum_{m'} \sum_{n'} w^{(l)}[m-m', n-n', c, d] h^{(l-1)}[m', n', c] \\ &\equiv w^{(l)}[m,n,c,d] * h^{(l-1)}[m,n,c]\end{aligned}$$

Back-prop, then, is also a kind of convolution, but with the filter flipped left-to-right and top-to-bottom. Flipped convolution is also known as "correlation."

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial h^{(l-1)}[m',n',c]} &= \sum_m \sum_n \sum_c w^{(l)}[m-m', n-n', c, d] \frac{d\mathcal{L}}{d\xi^{(l)}[m,n,d]} \\ &= w^{(l)}[-m', -n', c, d] * \frac{d\mathcal{L}}{d\xi^{(l)}[m',n',d]}\end{aligned}$$

Max Pooling

- Philosophy: the activation $h^{(l)}[m,n,c]$ should be greater than zero if the corresponding feature is detected anywhere within the vicinity of pixel (m,n). In fact, let's look for the *best matching* input pixel.
- Equation:

$$h^{(l)}[m,n,c] = \max_{m'=0}^{M-1} \max_{n'=0}^{M-1} \mathrm{ReLU}\left(\xi^{(l)}[mM+m', nM+n', c]\right)$$

where M is a max-pooling factor (often $M=2$, but not always).

Back-Prop for Max Pooling

The back-prop is pretty easy to understand. The activation gradient, $\frac{d\mathcal{L}}{dh^{(l)}[m,n,c]}$, is back-propagated to just one of the excitation gradients in its pool: the one that had the maximum value.

$$\frac{d\mathcal{L}}{d\xi^{(l)}[mM+m',nM+n',c]} = \begin{cases} \frac{d\mathcal{L}}{dh^{(l)}[m,n,c]} & \begin{array}{l} m'=m^*,\ \ n'=n^*, \\ h^{(l)}[m,n,c]>0, \end{array} \\ 0 & \text{otherwise,} \end{cases}$$

where

$$m^*,n^* = \operatorname*{argmax}_{m',n'} \xi^{(l)}[mM+m',nM+n',c],$$

Object Detection as Classification

Suppose that we are given a region of interest, $ROI = (x, y, w, h)$, and asked to decide whether the ROI is an object. We can do this by training a neural network to estimate the classifier output:

$$y_c(ROI) = \begin{cases} 1 & \text{ROI contains an object} \\ 0 & \text{ROI does not contain an object} \end{cases}$$

A neural net trained with MSE or CE will then compute

$$\hat{y}_c = \Pr\left(\text{ROI contains an object}\right)$$

Intersection over union (IOU)

We deal with partial-overlap by putting some sort of threshold on the intersection-over-union measure. Suppose the hypothesis is $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$, and the reference is $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$, then IOU is

$$IOU = \frac{I}{U} = \frac{\text{number of pixels in both ROI and REF}}{\text{number of pixels in either ROI or REF}},$$

where the intersection between REF and ROI is:

$$I = \left(\min\left(x_{REF} + w_{REF}, x_{ROI} + w_{ROI}\right) - \max\left(x_{REF}, x_{ROI}\right)\right) \times \\ \left(\min\left(y_{REF} + h_{REF}, y_{ROI} + h_{ROI}\right) - \max\left(y_{REF}, y_{ROI}\right)\right),$$

and their union is:

$$U = w_{REF}h_{REF} + w_{ROI}h_{ROI} - I$$

What pixels **should** be covered?

- The ROI is $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$.
- The anchor is (x_a, y_a, w_a, h_a).
- The true object is located at $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$.
- The regression target is:

$$\vec{y}_r = \begin{bmatrix} \frac{x_{REF}-x_a}{w_a} \\ \frac{y_{REF}-y_a}{h_a} \\ \ln\left(\frac{w_{REF}}{w_a}\right) \\ \ln\left(\frac{h_{REF}}{h_a}\right) \end{bmatrix}$$

Training a bbox regression network

The network is now trained with two different outputs, $\hat{y}_c$ and $\hat{y}_r$. The total loss is

$$\mathcal{L} = \mathcal{L}_c + \mathcal{L}_r$$

where $\mathcal{L}_c$ is BCE for the classifier output:

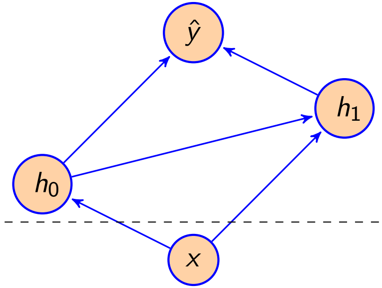
$$\mathcal{L}_c = -\frac{1}{n}\sum_{i=1}^{n}\left(y_{c,i}\ln \hat{y}_{c,i} + (1 - y_{c,i})\ln(1 - \hat{y}_{c,i})\right)$$

and $\mathcal{L}_r$ is zero if $y_c = 0$ (no object present), and MSE if $y_c = 1$:

$$\mathcal{L}_r = \frac{1}{2}\frac{\sum_{i=1}^{n} y_{c,i}\|\vec{y}_{r,i} - \hat{y}_{r,i}\|^2}{\sum_{i=1}^{n} y_{c,i}}$$

Outline

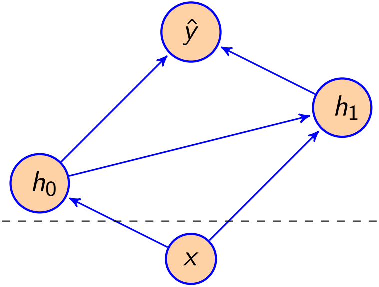
Flow Graphs



We often show the flow graph for the chain rule using bubbles and arrows, as shown above. You can imagine the chain rule as taking a summation along any cut through the flow graph—for example, the dashed line shown above. You take the total derivative from $\hat{y}$ to the cut, and then the partial derivative from there back to x.

$$\frac{d\hat{y}}{dx} = \sum_{i=0}^{N-1} \frac{d\hat{y}}{dh_i} \frac{\partial h_i}{\partial x}$$

Flow Graphs



$$\frac{d\hat{y}}{dx} = \sum_{i=0}^{N-1} \frac{d\hat{y}}{dh_i} \frac{\partial h_i}{\partial x}$$

For each h_i, we find the **total derivative** of $\hat{y}$ w.r.t. h_i, multiplied by the **partial derivative** of h_i w.r.t. x.

Recurrent Neural Net (RNN) = Nonlinear(IIR)

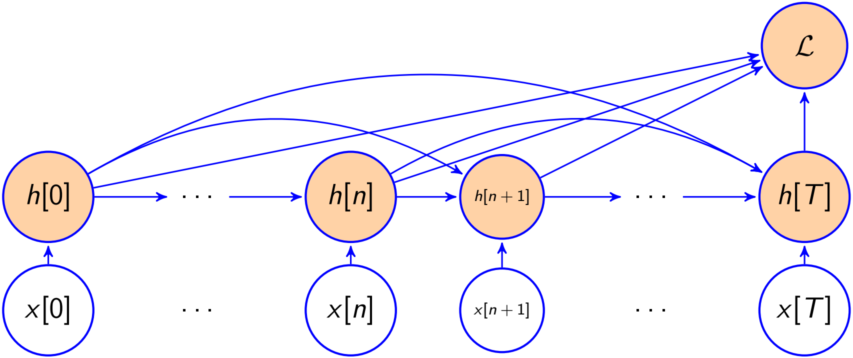
$$h[n] = \sigma\left(x[n] + \sum_{m=1}^{M-1} w[m]h[n-m]\right)$$

The coefficients, $w[m]$, are chosen to minimize the loss function. For example, suppose that the goal is to make $h[n]$ resemble a target signal $y[n]$; then we might use

$$\mathcal{L} = \frac{1}{2}\sum_{n=0}^{N}(h[n] - y[n])^2$$

and choose

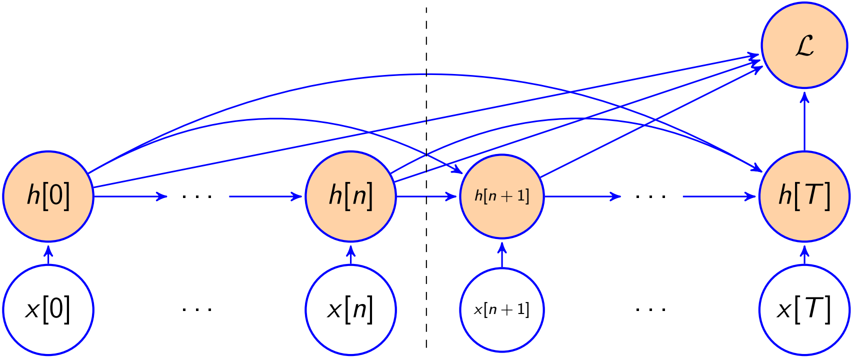
$$w[m] \leftarrow w[m] - \eta\frac{d\mathcal{L}}{dw[m]}$$



Here's a flow diagram that could represent:

$$h[n] = g\left(x[n] + \sum_{m=0}^{\infty} w[m]h[n-m]\right)$$

$$\mathcal{L} = \frac{1}{2}\sum_n (y[n] - h[n])^2$$



Back-propagation through time does this:

$$\frac{d\mathcal{L}}{dh[n]} = \frac{\partial \mathcal{L}}{\partial h[n]} + \sum_{m=1}^{T-n} \frac{d\mathcal{L}}{dh[n+m]} \frac{\partial h[n+m]}{\partial h[n]}$$

Outline

Long Short-Term Memory (LSTM)

The three gates are:

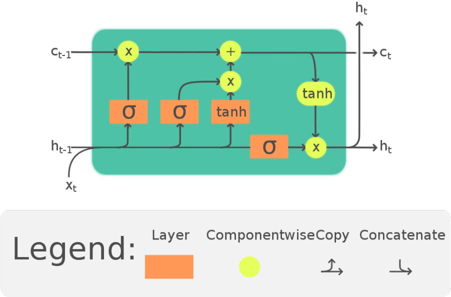
1. The cell remembers the past only when the forget gate is on, $f[t] = 1$.
2. The cell accepts input only when the input gate is on, $i[t] = 1$.

$$c[t] = f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$

3. The cell is output only when the output gate is on, $o[t] = 1$.

$$h[t] = o[t]c[t]$$

Neural Network Model: LSTM



$$i[t] = \text{input gate} = \sigma(w_i x[t] + u_i h[t-1] + b_i)$$

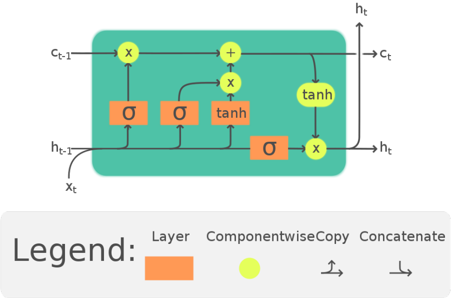
$$o[t] = \text{output gate} = \sigma(w_o x[t] + u_o h[t-1] + b_o)$$

$$f[t] = \text{forget gate} = \sigma(w_f x[t] + u_f h[t-1] + b_f)$$

$$c[t] = \text{memory cell} = f[t]c[t-1] + i[t]\tanh\left(w_c x[t] + u_c h[t-1] + b_c\right)$$

$$h[t] = \text{output} = o[t]c[t]$$

Back-Prop Through Time



$$\frac{d\mathcal{L}}{dh[t]} = \frac{\partial\mathcal{L}}{\partial h[t]} + \sum_{\xi\in\{i,o,f,c\}} \frac{d\mathcal{L}}{d\xi[t+1]} \frac{\partial\xi[t+1]}{\partial h[t]}$$

Back-Prop Through Time

Back-propagation for all of the other variables is easier, since only $c[t]$ has any direct connection from the current time to the next time:

$$\frac{d\mathcal{L}}{dc[t]} = \frac{d\mathcal{L}}{dh[t]}\frac{\partial h[t]}{\partial c[t]} + \frac{d\mathcal{L}}{dc[t+1]}\frac{\partial c[t+1]}{\partial c[t]}$$

$$\frac{d\mathcal{L}}{do[t]} = \frac{d\mathcal{L}}{dh[t]}\frac{\partial h[t]}{\partial o[t]}$$

$$\frac{d\mathcal{L}}{di[t]} = \frac{d\mathcal{L}}{dc[t]}\frac{\partial c[t]}{\partial i[t]}$$

$$\frac{d\mathcal{L}}{df[t]} = \frac{d\mathcal{L}}{dc[t]}\frac{\partial c[t]}{\partial f[t]}$$

Outline

Summary

- Neural networks & back-propagation
- CNN & Faster RCNN
- Partial derivatives & RNN
- LSTM