Deep Learning

Recurrent Networks:
Stability analysis and LSTMs
Which open source project?

```c
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */

static int indicate_policy(void)
{
    int error;
    if (fd == MARN_EPT) {
        /*
         * The kernel blank will coedl it to userspace.
         */
        if (ss->segment < mem_total)
            unblock_graph_and_set_blocked();
        else
            ret = 1;
        goto bail;
    }
    segaddr = in_SB(in.addr);
    selector = seg / 16;
    setup_works = true;
    for (i = 0; i < blocks; i++) {
        seq = buf[i++];
        bpf = bd->bd.next + i * search;
        if (fd) {
            current = blocked;
        }
    }
    rw->name = "Getjbbregs";
    bprm_self_clear1(&iv->version);
    regs->new = blocks[(BPF_STATS << info->historidac) | PFMR_CLOBATHINC_SECON]
    return segtable;
}
```
Related math. What is it talking about?

\textit{Proof.} Omitted.

\textbf{Lemma 0.1.} Let $C$ be a set of the construction.

Let $C$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{X}} = \mathcal{O}_{\mathcal{X}}(\mathcal{L})$$

\textit{Proof.} This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\text{étale}}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morph}_{\mathcal{X}} \times_{\mathcal{X}} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules.

\textbf{Lemma 0.2.} This is an integer $\mathbb{Z}$ is injective.

\textit{Proof.} See Spaces, Lemma ??.

\textbf{Lemma 0.3.} Let $\mathcal{S}$ be a scheme. Let $\mathcal{X}$ be a scheme and $\mathcal{X}$ is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let $\mathcal{X}$ be a scheme. Let $\mathcal{X}$ be a scheme which is equal to the formal complex.

The following to the construction of the lemma follows.

\textit{Proof.} Let $\mathcal{S}$ be a scheme covering. Let

$$\mathbf{b} : \mathcal{X} \to \mathcal{Y} : \mathcal{Y} \to \mathcal{Y} : \mathcal{Y} \to \mathcal{Y} \times_{\mathcal{X}} \mathcal{Y} \to \mathcal{X}.$$ be a morphism of algebraic spaces over $\mathcal{S}$ and $\mathcal{Y}$. 

\textit{Proof.} Let $\mathcal{X}$ be a nonzero scheme of $\mathcal{X}$. Let $\mathcal{X}$ be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_\mathcal{X}$-modules. The following are equivalent

\begin{enumerate}
    \item $\mathcal{F}$ is an algebraic space over $\mathcal{S}$.
    \item If $\mathcal{X}$ is an affine open covering.
\end{enumerate}

Consider a common structure on $\mathcal{X}$ and $\mathcal{X}$ the functor $\mathcal{O}_\mathcal{X}(U)$ which is locally of finite type.

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

\begin{align*}
    S & \rightarrow \mathcal{G} \\
    \downarrow & \downarrow \mathcal{G} \\
    \mathcal{G} & \rightarrow \mathcal{O}_{\mathcal{Y}} \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{Y}_s} \\
    \mathcal{O}_{\mathcal{X}} & \rightarrow \mathcal{O}_{\mathcal{X}}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s} \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L}) \\
    \mathcal{G}_s & \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{L})
\end{align*}

is a limit. Then $\mathcal{G}$ is a finite type and assume $\mathcal{S}$ is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type $\mathcal{F}_s$. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{\mathcal{X}_s}$ is a sheaf of rings.

\textit{Proof.} We have see that $X = \text{Spec}(\mathcal{R})$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of $\mathcal{X}$ is an open neighbourhood of $\mathcal{U}$.

\textit{Proof.} This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.

A reduced above we conclude that $\mathcal{U}$ is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field"

$$\mathcal{O}_{\mathcal{X}_s} \rightarrow \mathcal{F}_s : \mathcal{O}_{\mathcal{X}_s}(\mathcal{F}_s) \rightarrow \mathcal{O}_{\mathcal{X}_s}(\mathcal{O}_{\mathcal{X}_s}(\mathcal{F}_s))$$

is an isomorphism of covering of $\mathcal{O}_{\mathcal{X}_s}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that $\mathcal{X}$ is an isomorphism.

The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_{\mathcal{X}_s}$-algebra with $\mathcal{F}$ are opens of finite type over $\mathcal{S}$.

If $\mathcal{F}$ is a scheme theoretic image points.

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{\mathcal{X}_s}$ is a closed immersion, see Lemma ??, This is a sequence of $\mathcal{F}$ is a similar morphism.
Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servicious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be the [[Punjab Resolution]] (PJS)[http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a major tripad of aid exile.]]
The unreasonable effectiveness of recurrent neural networks..

• All previous examples were generated blindly by a recurrent neural network..

• http://karpathy.github.io/2015/05/21/rnn-effectiveness/

• Examples of models that analyze (or in this case, generate) time-series data
Story so far

• **Iterated structures** are good for analyzing time series data with short-time dependence on the past
  – These are “**Time delay**” neural nets, AKA **convnets**
Story so far

• Iterated structures are good for analyzing time series data with short-time dependence on the past
  – These are “Time delay” neural nets, AKA convnets

• **Recurrent structures** are good for analyzing time series data with long-term dependence on the past
  – These are recurrent neural networks
Recurrent structures can do what static structures cannot

- The addition problem: Add two N-bit numbers to produce a N+1-bit number
  - Input is binary
  - Will require large number of training instances
    - Output must be specified for every pair of inputs
    - Weights that generalize will make errors
  - Network trained for N-bit numbers will not work for N+1 bit numbers
MLPs vs RNNs

• The addition problem: Add two N-bit numbers to produce a N+1-bit number

• **RNN solution:** Very simple, can add two numbers of any size
MLP: The parity problem

• Is the number of “ones” even or odd
• Network must be complex to capture all patterns
  – XOR network, quite complex
  – Fixed input size
RNN: The parity problem

- Trivial solution
- Generalizes to input of any size
Story so far

• Recurrent structures can be trained by minimizing the divergence between the sequence of outputs and the sequence of desired outputs
  – Through gradient descent and backpropagation
Types of recursion

- Nothing special about a one step recursion
The behavior of recurrence..

- Returning to an old model..

\[ Y(t) = f(X(t - i), i = 1..K) \]

- When will the output “blow up”?
“BIBO” Stability

- Time-delay structures have bounded output if
  - The function $f()$ has bounded output for bounded input
    - Which is true of almost every activation function
      - $X(t)$ is bounded
  - “Bounded Input Bounded Output” stability
    - This is a highly desirable characteristic
Is this BIBO?

- Will this necessarily be BIBO?
Is this BIBO?

• Will this necessarily be BIBO?
  – Guaranteed if output and hidden activations are bounded
    • But will it saturate (and where)
  – What if the activations are linear?
Analyzing recurrence

• Sufficient to analyze the behavior of the hidden layer $h_k$ since it carries the relevant information
  – Will assume only a single hidden layer for simplicity
Analyzing Recursion

The streetlight effect is a type of observational bias where people only look for whatever they are searching by looking where it is easiest.

“I’m searching for my keys.”
Streetlight effect

• Easier to analyze linear systems
  – Will attempt to extrapolate to non-linear systems subsequently

• All activations are identity functions
  – \( z_k = W_h h_{k-1} + W_x x_k \), \( h_k = z_k \)
Linear systems

- \( h_k = W_h h_{k-1} + W_x x_k \)
  
  \(- h_{k-1} = W_h h_{k-2} + W_x x_{k-1} \)

- \( h_k = W_h^2 h_{k-2} + W_h W_x x_{k-1} + W_x x_k \)

- \( h_k = W_h^{k+1} h_{-1} + W_h^k W_x x_0 + W_h^{k-1} W_x x_1 + W_h^{k-2} W_x x_2 + \cdots \)

- \( h_k = H_k(h_{-1}) + H_k(x_0) + H_k(x_1) + H_k(x_2) + \cdots \)
  
  \(- = h_{-1} H_k(1_{-1}) + x_0 H_k(1_0) + x_1 H_k(1_1) + x_2 H_k(1_2) + \cdots \)

Where \( H_k(1_t) \) is the hidden response at time \( k \) when the input is \([0 \ 0 \ 0 \ \cdots \ 1 \ 0 \ \cdots \ 0]\) (where the 1 occurs in the \( t \)-th position) with 0 initial condition

- The initial condition may be viewed as an input of \( h_{-1} \) at \( t = -1 \)
Streetlight effect

- Sufficient to analyze the response to a single input at $t = 0$
  - Principle of superposition in linear systems:
    $$h_k = h_{-1}H_k(1_{-1}) + x_0H_k(1_0) + x_1H_k(1_1) + x_2H_k(1_2) + \cdots$$
Linear recursions

• Consider simple, scalar, linear recursion (note change of notation)
  \[ h(t) = wh(t-1) + cx(t) \]
  \[ h_0(t) = w^t cx(0) \]
  • Response to a single input at 0
Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  - $h(t) = Wh(t - 1) +Cx(t)$
  - $h_0(t) = W^tCx(0)$
    - Length of response vector to a single input at 0 is $|h_0(t)|$

- We can write $W = UΛU^{-1}$
  - $Wu_i = \lambda_i u_i$
  - For any vector $h$ we can write
    - $h = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
    - $Wh = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
    - $W^t h = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
  - $\lim_{t \to \infty} |W^t h| = a_m \lambda_m^t u_m$ where $m = \arg\max_j \lambda_j$
Linear recursions: Vector version

• Vector linear recursion (note change of notation)
  - \( h(t) = WH(t - 1) + Cx(t) \)
  - \( h_0(t) = W^t Cx(0) \)
    • Length of response vector to a single input at 0 is \( |h_0(t)| \)

• We can write \( W = U\Lambda U^{-1} \)
  - \( Wu_i = \lambda_i u_i \)

For any input, for large \( t \) the length of the hidden vector will expand or contract according to the \( t \)-th power of the largest eigen value of the hidden-layer weight matrix

- \( W^h = a_1\lambda_1 u_1 + a_2\lambda_2 u_2 + \cdots + a_n\lambda_n u_n \)
- \( \lim_{t \to \infty} |W^t h| = a_m\lambda_m u_m \) where \( m = \text{argmax}_j \lambda_j \)
Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  \[ h(t) = Wh(t - 1) + Cx(t) \]
  \[ h_0(t) = W^tCx(0) \]

- Length of response vector to a single input at 0 is \( |h_0(t)| \)

For any input, for large \( t \) the length of the hidden vector will expand or contract according to the \( t \) -th power of the largest eigen value of the hidden-layer weight matrix.

Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the second largest Eigen value..

And so on..

- \( Wh = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n \)
- \( W^t h = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n \)

\[ \lim_{t \to \infty} |W^t h| = a_m \lambda_m^t u_m \] where \( m = \arg\max_j \lambda_j \)
Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  \[ h(t) = Wl(t-\tau) + f(l(t)) \]

If \(|\lambda_{\text{max}}| > 1\) it will blow up, otherwise it will contract and shrink to 0 rapidly.

- Length of response vector to a single input at 0 is \(|h_0(t)|\)

For any input, for large \(t\) the length of the hidden vector will expand or contract according to the \(t\) th power of the largest eigen value of the hidden-layer weight matrix. Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the second largest Eigen value.

And so on...

- \(W^t h = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n\)
- \(\lim_{t \to \infty} |W^t h| = a_m \lambda_m^t u_m\) where \(m = \arg\max_j \lambda_j\)
Linear recursions: Vector version

What about at middling values of $t$? It will depend on the other eigen values of notation.

If $|\lambda_{\text{max}}| > 1$ it will blow up, otherwise it will contract and shrink to 0 rapidly.

For any input, for large $t$ the length of the hidden vector will expand or contract according to the $t$ th power of the largest eigen value of the hidden-layer weight matrix.

Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the second largest Eigen value.

And so on..

- $W^t h = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \ldots + a_n \lambda_n^t u_n$
- $\lim_{t \to \infty} |W^t h| = a_m \lambda_m^t u_m$ where $m = \arg\max_j \lambda_j$
Linear recursions

- Vector linear recursion
  - $h(t) = Wh(t - 1) + Cx(t)$
  - $h_0(t) = w^t cx(0)$
- Response to a single input $[1\ 1\ 1\ 1]$ at 0

\[
\lambda_{max} = 0.9
\]
\[|\lambda_{max}| = 1.1\]
\[|\lambda_{max}| = 1\]
\[\lambda_{max} = 1\]
Linear recursions

• Vector linear recursion
  
  $h(t) = Wh(t - 1) + Cx(t)$
  
  $h_0(t) = w^t cx(0)$

• Response to a single input $[1 1 1 1]$ at 0

Complex Eigenvalues

$\lambda_{\text{max}} = 0.9$

$\lambda_{2\text{nd}} = 0.5$

$\lambda_{2\text{nd}} = 0.1$

$\lambda_{\text{max}} = 1.1$

$\lambda_{\text{max}} = 1$

$\lambda_{\text{max}} = 1$
Lesson..

• In linear systems, long-term behavior depends entirely on the eigenvalues of the hidden-layer weights matrix
  – If the largest Eigen value is greater than 1, the system will “blow up”
  – If it is lesser than 1, the response will “vanish” very quickly
  – Complex Eigen values cause oscillatory response
    • Which we may or may not want
    • For smooth behavior, must force the weights matrix to have real Eigen values
      – Symmetric weight matrix
How about non-linearities (scalar)

\[ h(t) = f(wh(t - 1) + cx(t)) \]

- The behavior of scalar non-linearities
- **Left: Sigmoid, Middle: Tanh, Right: Relu**
  - Sigmoid: Saturates in a limited number of steps, regardless of \( w \)
  - Tanh: Sensitive to \( w \), but eventually saturates
    - “Prefers” weights close to 1.0
  - Relu: Sensitive to \( w \), can blow up
How about non-linearities (scalar)

\[ h(t) = f(wh(t - 1) + cx(t)) \]

• With a negative start
• Left: Sigmoid, Middle: Tanh, Right: Relu
  – Sigmoid: Saturates in a limited number of steps, regardless of \( w \)
  – Tanh: Sensitive to \( w \), but eventually saturates
  – Relu: For negative starts, has no response
Vector Process

\[ h(t) = f(W h(t - 1) + C x(t)) \]

- Assuming a uniform unit vector initialization
  - \([1, 1, 1, \ldots] / \sqrt{N}\)
  - Behavior similar to scalar recursion
- Eigenvalues less than 1.0 retain the most “memory”
Vector Process

\[ h(t) = f(W h(t - 1) + C x(t)) \]

- Assuming a uniform unit vector initialization
  - \([-1, -1, -1, ...] / \sqrt{N}\)
  - Behavior similar to scalar recursion

\begin{align*}
\text{sigmoid} & \quad \text{tanh} & \quad \text{relu}
\end{align*}
Stability Analysis

• Formal stability analysis considers convergence of “Lyapunov” functions
  – Alternately, Routh’s criterion and/or pole-zero analysis
  – Positive definite functions evaluated at $h$
  – Conclusions are similar: only the tanh activation gives us any reasonable behavior
    • And still has very short “memory”

• Lessons:
  – Bipolar activations (e.g. tanh) have the best memory behavior
  – Still sensitive to Eigenvalues of $W$
  – Best case memory is short
  – Exponential memory behavior
    • “Forgets” in exponential manner
How about deeper recursion

• Consider simple, scalar, linear recursion
  – Adding more “taps” adds more “modes” to memory in somewhat non-obvious ways

\[ h(t) = 0.5h(t - 1) + 0.25h(t - 5) + x(t) \]

\[ h(t) = 0.5h(t - 1) + 0.25h(t - 5) + 0.1h(t - 8) + x(t) \]
Stability Analysis

• Similar analysis of vector functions with non-linear activations is relatively straightforward
  
  – *Linear systems*: Routh’s criterion
    • And pole-zero analysis (involves tensors)
      – On board?
  
  – Non-linear systems: Lyapunov functions

• Conclusions do not change
Story so far

• Recurrent networks retain information from the infinite past in principle

• In practice, they tend to blow up or forget
  – If the largest Eigen value of the recurrent weights matrix is greater than 1, the network response may blow up
  – If its less than one, the response dies down very quickly

• The “memory” of the network also depends on the activation of the hidden units
  – Sigmoid activations saturate and the network becomes unable to retain new information
  – RELU activations blow up or vanish rapidly
  – Tanh activations are the most effective at storing memory
    • But still, for not very long
RNNs..

• Excellent models for time-series analysis tasks
  – Time-series prediction
  – Time-series classification
  – Sequence prediction..
  – They can even simplify problems that are difficult for MLPs

• But the memory isn’t all that great..
  – Also..
The vanishing gradient problem

• A particular problem with training deep networks..
  – (Any deep network, not just recurrent nets)
  – The gradient of the error with respect to weights is unstable..
Some useful preliminary math: The problem with training deep networks

- A multilayer perceptron is a nested function
  \[ Y = f_N \left( W_{N-1} f_{N-1} \left( W_{N-2} f_{N-2}(...W_0 X) \right) \right) \]
- \( W_k \) is the weights matrix at the \( k^{th} \) layer
- The error for \( X \) can be written as
  \[ Div(X) = D \left( f_N \left( W_{N-1} f_{N-1} \left( W_{N-2} f_{N-2}(...W_0 X) \right) \right) \right) \]
Training deep networks

• Vector derivative chain rule: for any $f(Wg(X))$:

$$\frac{df(Wg(X))}{dX} = \frac{df(Wg(X))}{dWg(X)} \frac{dWg(X)}{dg(X)} \frac{dg(X)}{dX}$$

Let $Z = Wg(X)$

$$\nabla_X f = \nabla_Z f . W . \nabla_X g$$

• Where

  – $\nabla_Z f$ is the jacobian matrix of $f(Z)$ w.r.t $Z$
    • Using the notation $\nabla_Z f$ instead of $J_f(z)$ for consistency
Training deep networks

- For

\[ \text{Div}(X) = D \left( f_N \left( W_{N-1} f_{N-1}(W_{N-2} f_{N-2}(\ldots W_0 X)) \right) \right) \]

- We get:

\[ \nabla_{f_k} \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \ldots \nabla f_{k+1} W_k \]

- Where
  - \( \nabla_{f_k} \text{Div} \) is the gradient \( \text{Div}(X) \) of the error w.r.t the output of the kth layer of the network
    - Needed to compute the gradient of the error w.r.t \( W_{k-1} \)
  - \( \nabla f_n \) is jacobian of \( f_N() \) w.r.t. to its current input
  - All blue terms are matrices
  - All function derivatives are w.r.t. the (entire, affine) argument of the function
Training deep networks

For
\[
\text{Div}(X) = D \left( f_N \left( W_{N-1} f_{N-1} \left( W_{N-2} f_{N-2} \left( \ldots W_0 X \right) \right) \right) \right)
\]

We get:
\[
\nabla_{f_k} \text{Div} = \nabla D \left( \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \ldots \nabla f_{k+1} \cdot W_k \right)
\]

Where
- \( \nabla_{f_k} \text{Div} \) is the gradient \( \text{Div}(X) \) of the error w.r.t the output of the kth layer of the network
  - Needed to compute the gradient of the error w.r.t \( W_{k-1} \)
- \( \nabla f_n \) is jacobian of \( f_N \) w.r.t. to its current input
- All blue terms are matrices

Lets consider these Jacobians for an RNN (or more generally for any network)
The Jacobian of the hidden layers for an RNN

\[ \nabla f_t(z_i) = \begin{bmatrix} f_{t,1}'(z_1) & 0 & \ldots & 0 \\ 0 & f_{t,2}'(z_2) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & f_{t,N}'(z_N) \end{bmatrix} \]

\[ h_i^{(1)}(t) = f_1 \left( z_i^{(1)}(t) \right) \]

- \( \nabla f_t() \) is the derivative of the output of the (layer of) hidden recurrent neurons with respect to their input
  - For vector activations: A full matrix
  - For scalar activations: A matrix where the diagonal entries are the derivatives of the activation of the recurrent hidden layer
The Jacobian

\[ h_i^{(1)}(t) = f_1 \left( z_i^{(1)}(t) \right) \]

- The derivative (or subgradient) of the activation function is always bounded
  - The diagonals (or singular values) of the Jacobian are bounded
- There is a limit on how much multiplying a vector by the Jacobian will scale it

\[
\nabla f_t(z_i) = \begin{bmatrix}
  f_{t,1}'(z_1) & 0 & \cdots & 0 \\
  0 & f_{t,2}'(z_2) & \cdots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \cdots & f_{t,N}'(z_N)
\end{bmatrix}
\]
The derivative of the hidden state activation

\[ \nabla f_t(z_i) = \begin{bmatrix} f_{t,1}'(z_1) & 0 & \ldots & 0 \\ 0 & f_{t,2}'(z_2) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & f_{t,N}'(z_N) \end{bmatrix} \]

- Most common activation functions, such as sigmoid, tanh() and RELU have derivatives that are always less than 1
- The most common activation for the hidden units in an RNN is the tanh()
  - The derivative of tanh() is never greater than 1 (and mostly less than 1)

- Multiplication by the Jacobian is always a *shrinking* operation
Training deep networks

• As we go back in layers, the Jacobians of the activations constantly *shrink* the derivative
  – After a few layers the derivative of the divergence at any time is totally “forgotten”
What about the weights

\[ \nabla f_k \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \cdots \nabla f_{k+1} \cdot W_k \]

• In a single-layer RNN, the weight matrices are identical
  - The conclusion below holds for any deep network, though

• The chain product for \( \nabla f_k \text{Div} \) will
  - Expand \( \nabla D \) along directions in which the singular values of
    the weight matrices are greater than 1
  - Shrink \( \nabla D \) in directions where the singular values are less
    than 1
  - Repeated multiplication by the weights matrix will result in
    Exploding or vanishing gradients
Exploding/Vanishing gradients

\[ \nabla_{f_k} \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \cdots \nabla f_{k+1} W_k \]

• Every blue term is a matrix

• \(\nabla D\) is proportional to the actual error
  – Particularly for \(L_2\) and KL divergence

• The chain product for \(\nabla_{f_k} \text{Div}\) will
  – \textit{Expand} \(\nabla D\) in directions where each stage has singular values greater than 1
  – \textit{Shrink} \(\nabla D\) in directions where each stage has singular values less than 1
Gradient problems in deep networks

- The gradients in the lower/earlier layers can *explode* or *vanish*
  - Resulting in insignificant or unstable gradient descent updates
  - Problem gets worse as network depth increases

\[
\nabla f_k \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \ldots \nabla f_{k+1} W_k
\]
Vanishing gradient examples..

• 19 layer MNIST model
  – Different activations: Exponential linear units, RELU, sigmoid, tanh
  – Each layer is 1024 units wide
  – Gradients shown at initialization
    • Will actually decrease with additional training

• Figure shows $\log|\nabla_{W_{\text{neuron}}} \text{Div}|$ where $W_{\text{neuron}}$ is the vector of incoming weights to each neuron
  – i.e. the gradient of the loss w.r.t. the entire set of weights to each neuron
Vanishing gradient examples..

- 19 layer MNIST model
  - Different activations: Exponential linear units, RELU, sigmoid, tanh
  - Each layer is 1024 units wide
  - Gradients shown at initialization
    - Will actually *decrease* with additional training
- Figure shows \( \log |\nabla W_{\text{neuron}} \text{Div}| \) where \( W_{\text{neuron}} \) is the vector of incoming weights to each neuron
  - I.e. the gradient of the loss w.r.t. the entire set of weights to each neuron
Vanishing gradient examples..

• 19 layer MNIST model
  – Different activations: Exponential linear units, RELU, sigmoid, tanh
  – Each layer is 1024 units wide
  – Gradients shown at initialization
    • Will actually decrease with additional training

• Figure shows $\log|\nabla W_{neuron} \cdot Div|$ where $W_{neuron}$ is the vector of incoming weights to each neuron
  – i.e. the gradient of the loss w.r.t. the entire set of weights to each neuron
Vanishing gradient examples..

19 layer MNIST model
- Different activations: Exponential linear units, RELU, sigmoid, tanh
- Each layer is 1024 units wide
- Gradients shown at initialization
  - Will actually decrease with additional training

Figure shows \( \log |\nabla W_{\text{neuron}} \cdot \text{Div}| \) where \( W_{\text{neuron}} \) is the vector of incoming weights to each neuron
- i.e. the gradient of the loss w.r.t. the entire set of weights to each neuron
Vanishing gradient examples..

ELU activation, Individual instances

• 19 layer MNIST model
  – Different activations: Exponential linear units, RELU, sigmoid, tanh
  – Each layer is 1024 units wide
  – Gradients shown at initialization
    • Will actually decrease with additional training

• Figure shows $\log |\nabla_{W_{neuron}} \text{Div}|$ where $W_{neuron}$ is the vector of incoming weights to each neuron
  – I.e. the gradient of the loss w.r.t. the entire set of weights to each neuron
Vanishing gradients

• ELU activations maintain gradients longest
• But in all cases gradients effectively vanish after about 10 layers!
  – Your results may vary

• Both batch gradients and gradients for individual instances disappear
  – In reality a tiny number will actually blow up.
Story so far

- Recurrent networks retain information from the infinite past in principle

- In practice, they are poor at memorization
  - The hidden outputs can blow up, or shrink to zero depending on the Eigen values of the recurrent weights matrix
  - The memory is also a function of the activation of the hidden units
    - Tanh activations are the most effective at retaining memory, but even they don’t hold it very long

- Deep networks also suffer from a “vanishing or exploding gradient” problem
  - The gradient of the error at the output gets concentrated into a small number of parameters in the earlier layers, and goes to zero for others
Recurrent nets are very deep nets

\[ \nabla f_k \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \cdots \nabla f_{k+1} W_k \]

- The relation between \( X(0) \) and \( Y(T) \) is one of a very deep network
  - Gradients from errors at \( t = T \) will vanish by the time they’re propagated to \( t = 0 \)
Recall: Vanishing stuff..

- Stuff gets forgotten in the forward pass too
  - Each weights matrix and activation can shrink components of the input
The long-term dependency problem

- Any other pattern of any length can happen between pattern 1 and pattern 2
  - RNN will “forget” pattern 1 if intermediate stuff is too long
  - “Jane” → the next pronoun referring to her will be “she”
- Must know to “remember” for extended periods of time and “recall” when necessary
  - Can be performed with a multi-tap recursion, but how many taps?
  - Need an alternate way to “remember” stuff

Jane had a quick lunch in the bistro. Then she..
And now we enter the domain of..
Exploding/Vanishing gradients

\( \nabla f_k \text{Div} = \nabla D \cdot \nabla f_N \cdot W_{N-1} \cdot \nabla f_{N-1} \cdot W_{N-2} \ldots \nabla f_{k+1} W_k \)

- Can we replace this with something that doesn’t fade or blow up?

- Can we have a network that just “remembers” arbitrarily long, to be recalled on demand?
  - Not be directly dependent on vagaries of network parameters, but rather on input-based determination of whether it must be remembered
    - Retain memoris until a switch based on the input flags them as ok to forget
      - Or remember less

- \( \text{Memory}(k) \approx C \sigma_k C \sigma_{k-1} C \ldots \sigma_1 \)

- \( \nabla f_k \text{Div} \approx \nabla D C \sigma_N^' C \sigma_{N-1}^' C \ldots \sigma_k^' \)
Enter – the constant error carousel

- History is carried through uncompressed
  - No weights, no nonlinearities
  - Only scaling is through the $\sigma$ “gating” term that captures other triggers
  - E.g. “Have I seen Pattern2”? 
Enter – the constant error carousel

• Actual non-linear work is done by other portions of the network
  – Neurons that compute the workable state from the memory
Enter – the constant error carousel

The gate $\sigma$ depends on current input, current hidden state...

• The gate $\sigma$ depends on current input, current hidden state...
Enter – the constant error carousel

- The gate $\sigma$ depends on current input, current hidden state... and other stuff...
Enter – the constant error carousel

- The gate $\sigma$ depends on current input, current hidden state... and other stuff...
- Including, obviously, what is currently in raw memory
Enter the LSTM

- *Long Short-Term Memory*
- Explicitly latch information to prevent decay / blowup

- Following notes borrow liberally from
- [http://colah.github.io/posts/2015-08-Understanding-LSTMs/](http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
Standard RNN

- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a $\tanh()$ activation function
  - As mentioned earlier, $\tanh()$ is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant
• The $\sigma()$ are *multiplicative gates* that decide if something is important or not

• Remember, every line actually represents a *vector*
LSTM: Constant Error Carousel

- Key component: a *remembered cell state*
LSTM: CEC

- $C_t$ is the linear history carried by the constant-error carousel
- Carries information through, only affected by a gate
  - And addition of history, which too is gated..
LSTM: Gates

- Gates are simple sigmoidal units with outputs in the range (0, 1)
- Controls how much of the information is to be let through
**LSTM: Forget gate**

The first gate determines whether to carry over the history or to forget it.

- More precisely, how much of the history to carry over.
- Also called the “forget” gate.
- Note, we’re actually distinguishing between the cell memory $C$ and the state $h$ that is coming over time! They’re related though.

$ f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f) $
LSTM: Input gate

- The second input has two parts
  - A perceptron layer that determines if there’s something new and interesting in the input
  - A gate that decides if it’s worth remembering

\[
i_t = \sigma \left( W_i \cdot [h_{t-1}, x_t] + b_i \right)
\]

\[
\tilde{C}_t = \tanh \left( W_C \cdot [h_{t-1}, x_t] + b_C \right)
\]
The second input has two parts

- A perceptron layer that determines if there’s something interesting in the input
- A gate that decides if it’s worth remembering
- If so, it’s added to the current memory cell

$$C_t = f_t \ast C_{t-1} + i_t \ast \tilde{C}_t$$
LSTM: Output and Output gate

The **output** of the cell

- Simply compress it with tanh to make it lie between 1 and -1
  - Note that this compression no longer affects our ability to **carry** memory forward
- Controlled by an **output** gate
  - To decide if the memory contents are worth reporting at **this** time

\[
o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)
\]
\[
h_t = o_t \times \tanh(C_t)
\]
LSTM: The “Peephole” Connection

- The raw memory is informative by itself and can also be input
  - Note, we’re using both $C$ and $h$

\[
\begin{align*}
    f_t &= \sigma \left( W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f \right) \\
    i_t &= \sigma \left( W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i \right) \\
    o_t &= \sigma \left( W_o \cdot [C_t, h_{t-1}, x_t] + b_o \right)
\end{align*}
\]
The complete LSTM unit

- With input, output, and forget gates and the peephole connection.
Backpropagation rules: Forward

- Forward rules:

  **Gates**
  \[
  f_t = \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \\
  i_t = \sigma(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \\
  o_t = \sigma(W_o \cdot [C_t, h_{t-1}, x_t] + b_o)
  \]

  **Variables**
  \[
  \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
  C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\
  h_t = o_t \cdot \tanh(C_t)
  \]
Notes on the pseudocode

Class LSTM_cell

• We will assume an object-oriented program
• Each LSTM unit is assumed to be an “LSTM cell”
• There’s a new copy of the LSTM cell at each time, at each layer
• LSTM cells retain local variables that are not relevant to the computation outside the cell
  – These are static and retain their value once computed, unless overwritten
LSTM cell (single unit)  
Definitions

# Input:
#   C : current value of CEC
#   h : Current hidden state value ("output" of cell)
#   x:  Current input
#   [W,b]: The set of all model parameters for the cell
#       These include all weights and biases
# Output
#   C : Next value of CEC
#   h : Next value of h
# In the function: sigmoid(x) = 1/(1+exp(-x))
#                 performed component-wise

# Static local variables to the cell
static local \( z_f, z_i, z_c, z_o, f, i, o, C_i \)
function \([C,h] = \text{LSTM	extunderscore cell.forward}(C,h,x,[W,b])\)
   code on next slide
Continuing from previous slide

Note: \([W,h]\) is a set of parameters, whose individual elements are shown in red within the code. These are passed in.

Static local variables which aren’t required outside this cell

```python
static local z_f, z_i, z_c, z_o, f, i, o, C_i

function [C_o, h_o] = LSTM_cell.forward(C, h, x, [W, h])

z_f = W_{fc}C + W_{fh}h + W_{fx}x + b_f
f = sigmoid(z_f) # forget gate

z_i = W_{ic}C + W_{ih}h + W_{ix}x + b_i
i = sigmoid(z_i) # input gate

z_c = W_{cc}C + W_{ch}h + W_{cx}x + b_c
C_i = tanh(z_c) # Detecting input pattern

C_o = f o C + i o C_i # “o” is component-wise multiply

z_o = W_{oc}C_o + W_{oh}h + W_{ox}x + b_o
o = sigmoid(z_o) # output gate

h_o = o o tanh(C) # “o” is component-wise multiply

return C_o, h_o
```
LSTM network forward

# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#             for the lth hidden layer
# W_o and b_o are output layer weights and biases

for t = 0:T-1  # Including both ends of the index
    h(t,0) = x(t)  # Vectors. Initialize h(0) to input
    for l = 1:L  # hidden layers operate at time t
        [C(t,l),h(t,l)] = LSTM_cell(t,l).forward(...
        ...C(t-1,l),h(t-1,l),h(t,l-1) [W{l},b{l}])
    z_o(t) = W_o h(t,L) + b_o
    Y(t) = softmax( z_o(t) )
Backpropagation rules: Backward

\[ \nabla_{C_t} Div = \]

\[ C_{t-1} \rightarrow C_t \rightarrow C_{t+1} \]

\[ h_{t-1} \rightarrow h_t \rightarrow h_{t+1} \]

\[ x_t \rightarrow \sigma() \rightarrow i_t \rightarrow \sigma() \rightarrow C_t \rightarrow \tanh \rightarrow o_t \rightarrow \sigma() \rightarrow z_t \rightarrow C_t \rightarrow \tanh \rightarrow C_{t+1} \rightarrow x_{t+1} \rightarrow \sigma() \rightarrow \hat{C}_{t+1} \]
Backpropagation rules: Backward

\[ \nabla_{c_t} Div = \nabla_{h_t} Div \circ o_t \circ \tanh'(.) \]
Backpropagation rules: Backward

\[ \nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ \text{tanh}'(.) + \text{tanh}(.) \circ \sigma'(.) W_{co}) \]
Backpropagation rules: Backward

\[ \nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.) W_{Co}) + \nabla_{C_{t+1}} Div \circ f_{t+1} + \]
Backpropagation rules: Backward

\[ \nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.) W_{Co}) + \\
\nabla_{C_{t+1}} Div \circ (f_{t+1} + C_t \circ \sigma'(.) W_{Cf}) \]
Backpropagation rules: Backward

\[
\nabla_{C_t} \text{Div} = \nabla_{h_t} \text{Div} \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.) W_{Co}) + \\
\nabla_{C_{t+1}} \text{Div} \circ (f_{t+1} + C_t \circ \sigma'(.) W_{Cf} + \tilde{C}_{t+1} \circ \sigma'(.) W_{Ci})
\]
Backpropagation rules: Backward

\[ \nabla_{C_t} D_{iv} = \nabla_{h_t} D_{iv} \odot \left( o_t \odot \tanh'(.) + \tanh(.) \odot \sigma'(.) W_{Co} \right) + \\
\nabla_{C_{t+1}} D_{iv} \odot \left( f_{t+1} + C_t \odot \sigma'(.) W_{cf} + \tilde{C}_{t+1} \odot \sigma'(.) W_{ci} \right) \]

\[ \nabla_{h_t} D_{iv} = \nabla_{z_t} D_{iv} \nabla_{h_t} z_t \]
Backpropagation rules: Backward

\[ \nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.)W_{Co}) + \\
\nabla_{C_{t+1}} Div \circ (f_{t+1} + C_t \circ \sigma'(.)W_{Cf} + \tilde{C}_{t+1} \circ \sigma'(.)W_{Ci}) \]

\[ \nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{C_{t+1}} Div \circ C_t \circ \sigma'(.)W_{hf} \]

94
Backpropagation rules: Backward

\[ \nabla_{c_t} Div = \nabla_{h_t} Div \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.)W_{co}) + \\
\nabla_{c_{t+1}} Div \circ (f_{t+1} + c_t \circ \sigma'(.)W_{cf} + \tilde{c}_{t+1} \circ \sigma'(.)W_{ci}) \\
\]

\[ \nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{c_{t+1}} Div \circ (c_t \circ \sigma'(.)W_{hf} + \tilde{c}_{t+1} \circ \sigma'(.)W_{hi}) \]
Backpropagation rules: Backward

\[ \nabla_{c_t} Div = \nabla_{h_t} Div \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.)W_{co}) + \\
\nabla_{c_{t+1}} Div \circ (f_{t+1} + c_t \circ \sigma'(.)W_{cf} + \tilde{c}_{t+1} \circ \sigma'(.)W_{ci}) \]

\[ \nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{c_{t+1}} Div \circ (c_t \circ \sigma'(.)W_{hf} + \tilde{c}_{t+1} \circ \sigma'(.)W_{hi}) + \\
\nabla_{c_{t+1}} Div \circ i_{t+1} \circ \tanh'(.)W_{hi} \]
Backpropagation rules: Backward

\[ \nabla C_t \text{Div} = \nabla h_t \text{Div} \circ (o_t \circ \tanh'(.) + \tanh(.) \circ \sigma'(.) W_{C_o}) + \nabla C_{t+1} \text{Div} \circ (f_{t+1} + C_t \circ \sigma'(.) W_{C_f} + \tilde{C}_{t+1} \circ \sigma'(.) W_{C_i}) \]

\[ \nabla h_t \text{Div} = \nabla z_t \text{Div} \nabla h_t z_t + \nabla C_{t+1} \text{Div} \circ (C_t \circ \sigma'(.) W_{h_f} + \tilde{C}_{t+1} \circ \sigma'(.) W_{h_i}) + \nabla C_{t+1} \text{Div} \circ o_{t+1} \circ \tanh'(.) W_{h_i} + \nabla h_{t+1} \text{Div} \circ \tanh(.) \circ \sigma'(.) W_{h_o} \]
Backpropagation rules: Backward

Not explicitly deriving the derivatives w.r.t weights; Left as an exercise

$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ \text{tanh}'(.) W_{Ch} + \text{tanh}(.) \circ \sigma'(.) W_{Co}) +$$
$$\nabla_{h_t} C_{t+1} \circ (f_{t+1} + C_t \circ \sigma'(.) W_{Cf} + \tilde{C}_{t+1} \circ \sigma'(.) W_{Ci})$$

$$\nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{C_{t+1}} Div \circ (C_t \circ \sigma'(.) W_{hf} + \tilde{C}_{t+1} \circ \sigma'(.) W_{hi}) +$$
$$\nabla_{C_{t+1}} Div \circ o_{t+1} \circ \text{tanh}'(.) W_{hi} + \nabla_{h_{t+1}} Div \circ \text{tanh}(.) \circ \sigma'(.) W_{ho}$$
Notes on the backward pseudocode

Class LSTM_cell

• We first provide backward computation within a cell
• For the backward code, we will assume the static variables computed during the forward are still available
• The following slides first show the forward code for reference
• Subsequently we will give you the backward, and explicitly indicate which of the forward equations each backward equation refers to
  – The backward code for a cell is long (but simple) and extends over multiple slides
# Continuing from previous slide

# Note: \([W,h]\) is a set of parameters, whose individual elements are shown in red within the code. These are passed in

# Static local variables which aren’t required outside this cell
static local \(z_f, z_i, z_c, z_o, f, i, o, C_i\)

function \([C_o, h_o] = \text{LSTM\_cell\_forward}(C, h, x, [W, h])\)

\[
z_f = W_{fc}C + W_{fh}h + W_{fx}x + b_f
\]

\(f = \text{sigmoid}(z_f)\) # forget gate

\[
z_i = W_{ic}C + W_{ih}h + W_{ix}x + b_i
\]

\(i = \text{sigmoid}(z_i)\) # input gate

\[
z_c = W_{cc}C + W_{ch}h + W_{cx}x + b_c
\]

\(C_i = \text{tanh}(z_c)\) # Detecting input pattern

\(C_o = f \circ C + i \circ C_i\) # “\(\circ\)” is component-wise multiply

\[
z_o = W_{oc}C_o + W_{oh}h + W_{ox}x + b_o
\]

\(o = \text{sigmoid}(z_o)\) # output gate

\(h_o = o \circ \text{tanh}(C)\) # “\(\circ\)” is component-wise multiply

return \(C_o, h_o\)
# Static local variables carried over from forward

```plaintext
static local z_f, z_i, z_c, z_o, f, i, o, C_i
```

```plaintext
function [dC, dh, dx, d[W, b]] = LSTM_cell.backward(dC_o, dh_o, C, h, C_o, h_o, [W, b])
    # First invert h_o = o ∘ tanh(C)
    do = dh_o ∘ tanh(C_o)^T
    d tanhC_o = dh_o ∘ o
    dC_o += dtanhC_o ∘ (1-tanh^2(C_o))^T  #(1-tanh^2) is the derivative of tanh

    # Next invert o = sigmoid(z_o)
    dz_o = do ∘ sigmoid(z_o)^T ∘ (1-sigmoid(z_o))^T  # do x derivative of sigmoid(z_o)

    # Next invert z_o = W_oc*C_o + W_oh*h + W_ox*x + b_o
    dC_o += dz_o * W_oc  # Note - this is a regular matrix multiply
    dh = dz_o * W_oh
    dx = dz_o * W_ox

    dW_oc = C_o * dz_o  # Note - this multiplies a column vector by a row vector
    dW_oh = h * dz_o
    dW_ox = x * dz_o
    db_o = dz_o

    # Next invert C_o = f ∘ C + i ∘ C_i
    dC = dC_o ∘ f
    dC_i = dC_o ∘ i
    di = dC_o ∘ C_i
    df = dC_o ∘ C
```
# Next invert $C_i = \tanh(z_c)$
$d z_c = d C_i \circ (1 - \tanh^2(z_c))^T$

# Next invert $z_c = W_{cc} C + W_{ch} h + W_{cx} x + b_c$
$d C = dz_c W_{cc}$
$d h = dz_c W_{ch}$
$d x = dz_c W_{cx}$

$d W_{cc} = C \ dz_c$,
$d W_{ch} = h \ dz_c$,
$d W_{cx} = x \ dz_c$,
$db_c = dz_c$

# Next invert $i = \text{sigmoid}(z_i)$
$d z_i = d i \circ \text{sigmoid}(z_i)^T \circ (1 - \text{sigmoid}(z_i))^T$

# Next invert $z_i = W_{ic} C + W_{ih} h + W_{ix} x + b_i$
$d C = dz_i W_{ic}$
$d h = dz_i W_{ih}$
$d x = dz_i W_{ix}$

$d W_{ic} = C \ dz_i$
$d W_{ih} = h \ dz_i$
$d W_{ix} = x \ dz_i$
$db_i = dz_i$
LSTM cell backward (continued)

# Next invert \( f = \text{sigmoid}(z_f) \)
\[
dz_f = df \circ \text{sigmoid}(z_f)^T \circ (1 - \text{sigmoid}(z_f))^T
\]

# Finally invert \( z_f = W_{fc} C + W_{fh} h + W_{fx} x + b_f \)
\[
dC += dz_f W_{fc}
dh += dz_f W_{fh}
dx += dz_f W_{fx}
\]
\[
dW_{fc} = C \ dz_f
dW_{fh} = h \ dz_f
dW_{fx} = x \ dz_f
db_f = dz_f
\]

return dC, dh, dx, d[W, b]

# d[W,b] is shorthand for the complete set of weight and bias derivatives
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
# for the l^{th} hidden layer
# W_o and b_o are output layer weights and biases

for t = 0:T-1  # Including both ends of the index
    h(t,0) = x(t)  # Vectors. Initialize h(0) to input
    for l = 1:L  # hidden layers operate at time t
        [C(t,l),h(t,l)] = LSTM_cell(t,l).forward(…
                        …C(t-1,l),h(t-1,l),h(t,l-1)[W{l},b{l}])
    z_o(t) = W_o h(t,L) + b_o
    Y(t) = softmax( z_o(t) )
LSTM network backward

# Assuming \( h(-1,*) \) is known and \( C(-1,*)=0 \)
# Assuming \( L \) hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# \([W\{l\},b\{l\}]\) are the entire set of weights and biases
#             for the \( l \)th hidden layer
# \( W_\circ \) and \( b_\circ \) are output layer weights and biases
# \( Y \) is the output of the network
# Assuming \( dW_\circ \) and \( db_\circ \) and \( d[W\{l\} \ b\{l\}] \) (for all \( l \)) are
# all initialized to 0 at the start of the computation

for \( t = T-1:0 \)  # Including both ends of the index
    \( dz_\circ = dY(t) \odot \text{sigmoid}(z_\circ(t))^T \odot (1 - \text{sigmoid}(z_\circ(t)))^T \)
    \( dW_\circ += h(t,L) \ dz_\circ(t) \)
    \( dh(t,L) = dz_\circ(t)W_\circ \)
    \( db_\circ += dz_\circ(t) \)

for \( l = L-1:0 \)
    \[ dC(t,l), dh(t,l), dx(t,l), d[W,b] \] = 
    \[ \ldots \] LSTM\_cell\( (t,l) \).backward(\ldots
    \[ dC(t+1,l), \ dh(t+1,l), dx(t,l+1), C(t,l), h(t,l), \ldots \]
    \[ \ldots C(t,l), h(t,l), [W\{l\},b\{l\}] ] \)
    \( d[W\{l\} \ b\{l\}] += d[W,b] \)
Gated Recurrent Units: Let's simplify the LSTM

- Simplified LSTM which addresses some of your concerns of why

\[ z_t = \sigma (W_z \cdot [h_{t-1}, x_t]) \]
\[ r_t = \sigma (W_r \cdot [h_{t-1}, x_t]) \]
\[ \tilde{h}_t = \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \]
\[ h_t = (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t \]
Gated Recurrent Units: Let's simplify the LSTM

• Combine forget and input gates
  – In new input is to be remembered, then this means old memory is to be forgotten
    • Why compute twice?

\[
\begin{align*}
  z_t &= \sigma (W_z \cdot [h_{t-1}, x_t]) \\
  r_t &= \sigma (W_r \cdot [h_{t-1}, x_t]) \\
  \tilde{h}_t &= \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \\
  h_t &= (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t
\end{align*}
\]
Gated Recurrent Units: Let’s simplify the LSTM

\[
\begin{align*}
  z_t &= \sigma (W_z \cdot [h_{t-1}, x_t]) \\
  r_t &= \sigma (W_r \cdot [h_{t-1}, x_t]) \\
  \tilde{h}_t &= \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \\
  h_t &= (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t
\end{align*}
\]

- Don’t bother to separately maintain compressed and regular memories
  - Pointless computation!
  - Redundant representation
LSTM Equations

- \( i \): input gate, how much of the new information will be let through the memory cell.
- \( f \): forget gate, responsible for information that should be thrown away from memory cell.
- \( o \): output gate, how much of the information will be passed to expose to the next time step.
- \( g \): self-recurrent which is equal to standard RNN
- \( c_t \): internal memory of the memory cell
- \( s_t \): hidden state
- \( y \): final output

\[
\begin{align*}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o \\
y &= \text{softmax}(Vs_t)
\end{align*}
\]
LSTM architectures example

- Each green box is now an entire LSTM or GRU unit
- Also keep in mind each box is an array of units
Bidirectional LSTM

- Like the BRNN, but now the hidden nodes are LSTM units.
- Can have multiple layers of LSTM units in either direction
  - It's also possible to have MLP feed-forward layers between the hidden layers.
- The output nodes (orange boxes) may be complete MLPs.
Story so far

• Recurrent networks are poor at memorization
  – Memory can explode or vanish depending on the weights and activation

• They also suffer from the vanishing gradient problem during training
  – Error at any time cannot affect parameter updates in the too-distant past
  – E.g. seeing a “close bracket” cannot affect its ability to predict an “open bracket” if it happened too long ago in the input

• LSTMs are an alternative formalism where memory is made more directly dependent on the input, rather than network parameters/structure
  – Through a “Constant Error Carousel” memory structure with no weights or activations, but instead direct switching and “increment/decrement” from pattern recognizers
  – Do not suffer from a vanishing gradient problem but do suffer from exploding gradient issue
Significant issues

• The Divergence
• How to use these nets..
• This and more in next couple of classes..