CSC413/2516 Lecture 7: Generalization & Recurrent Neural Networks

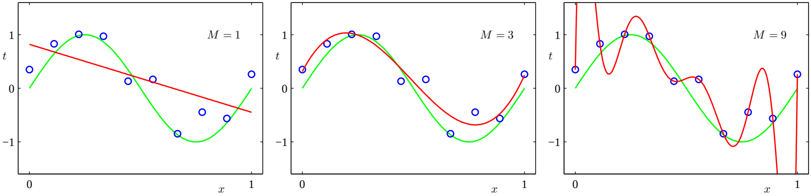
Jimmy Ba

Overview

- We've focused so far on how to *optimize* neural nets — how to get them to make good predictions on the training set.
- How do we make sure they generalize to data they haven't seen before?
- Even though the topic is well studied, it's still poorly understood.

Generalization

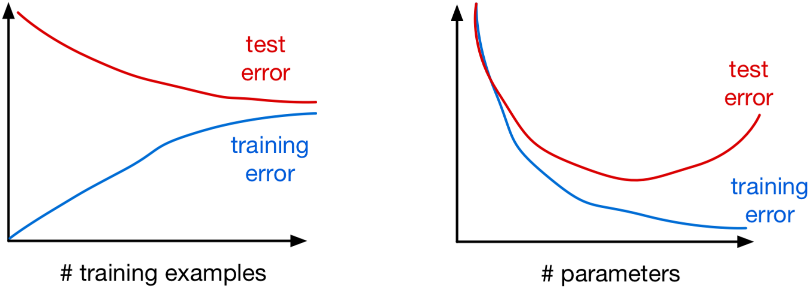
Recall: overfitting and underfitting



We'd like to minimize the generalization error, i.e. error on novel examples.

Generalization

- Training and test error as a function of # training examples and # parameters:



Our Bag of Tricks

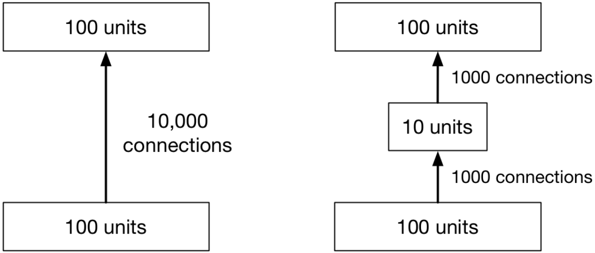
- How can we train a model that's complex enough to model the structure in the data, but prevent it from overfitting? I.e., how to achieve low bias and low variance?
- Our bag of tricks
  - data augmentation
  - reduce the number of paramters
  - weight decay
  - early stopping
  - ensembles (combine predictions of different models)
  - stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks.

Data Augmentation

- The best way to improve generalization is to collect more data!
- Suppose we already have all the data we're willing to collect. We can augment the training data by transforming the examples. This is called data augmentation.
- Examples (for visual recognition)
    - translation
    - horizontal or vertical flip
    - rotation
    - smooth warping
    - noise (e.g. flip random pixels)
- Only warp the training, not the test, examples.
- The choice of transformations depends on the task. (E.g. horizontal flip for object recognition, but not handwritten digit recognition.)

Reducing the Number of Parameters

- Can reduce the number of layers or the number of paramters per layer.
- Adding a linear bottleneck layer is another way to reduce the number of parameters:



- The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
- Remember how linear layers don't make a network more expressive? They might still improve generalization.

Weight Decay

- We've already seen that we can regularize a network by penalizing large weight values, thereby encouraging the weights to be small in magnitude.

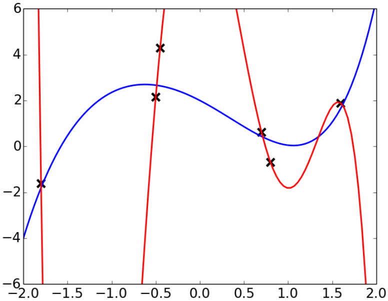
$$\mathcal{J}_{\mathrm{reg}} = \mathcal{J} + \lambda \mathcal{R} = \mathcal{J} + \frac{\lambda}{2} \sum_j w_j^2$$

- We saw that the gradient descent update can be interpreted as weight decay:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}\end{aligned}$$

Weight Decay

Why we want weights to be small:



$$y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2$$

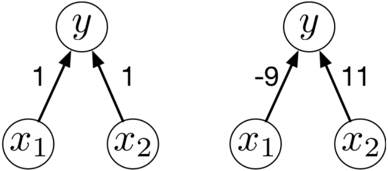
$$y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12$$

The red polynomial overfits. Notice it has really large coefficients.

Weight Decay

Why we want weights to be small:

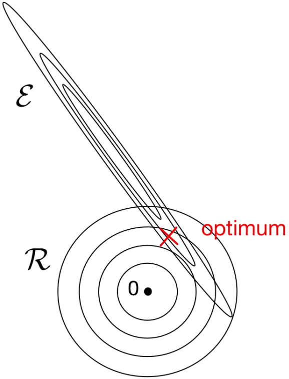
- Suppose inputs x_1 and x_2 are nearly identical. The following two networks make nearly the same predictions:



- But the second network might make weird predictions if the test distribution is slightly different (e.g. x_1 and x_2 match less closely).

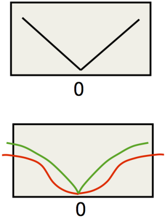
Weight Decay

- The geometric picture:

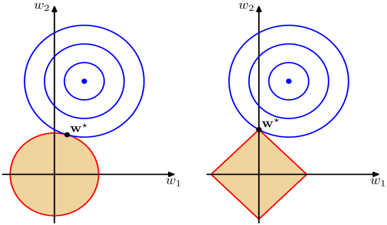


Weight Decay

- There are other kinds of regularizers which encourage weights to be small, e.g. sum of the absolute values.
- These alternative penalties are commonly used in other areas of machine learning, but less commonly for neural nets.
- Regularizers differ by how strongly they prioritize making weights exactly zero, vs. not being very large.



— Hinton, Coursera lectures



L2 regularization

$$\mathcal{R} = \sum_i w_i^2$$

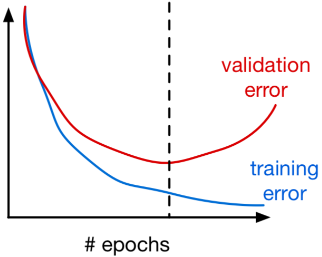
L1 regularization

$$\mathcal{R} = \sum_i |w_i|$$

— Bishop, *Pattern Recognition and Machine Learning*

Early Stopping

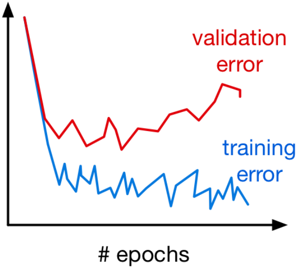
- We don't always want to find a global (or even local) optimum of our cost function. It may be advantageous to stop training early.



- Early stopping: monitor performance on a validation set, stop training when the validtion error starts going up.

Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.



- Determining when the validation error has actually leveled off can be tricky.

Early Stopping

- Why does early stopping work?
  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.
  - If you are using sigmoidal units, and the weights start out small, then the inputs to the activation functions take only a small range of values.
    - Therefore, the network starts out approximately linear, and gradually becomes more nonlinear (and hence more powerful).

Ensembles

- If a loss function is convex (with respect to the predictions), you have a bunch of predictions, and you don't know which one is best, you are always better off averaging them.

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1$$

- This is true no matter where they came from (trained neural net, random guessing, etc.). Note that only the loss function needs to be convex, not the optimization problem.
- Examples: squared error, cross-entropy, hinge loss
- If you have multiple candidate models and don't know which one is the best, maybe you should just average their predictions on the test data. The set of models is called an ensemble.
- Averaging often helps even when the loss is nonconvex (e.g. 0–1 loss).

Ensembles

- Some examples of ensembles:
  - Train networks starting from different random initializations. But this might not give enough diversity to be useful.
  - Train networks on differnet subsets of the training data. This is called bagging.
  - Train networks with different architectures or hyperparameters, or even use other algorithms which aren't neural nets.
- Ensembles can improve generalization quite a bit, and the winning systems for most machine learning benchmarks are ensembles.
- But they are expensive, and the predictions can be hard to interpret.

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units (i.e. sets their activations to zero).

$$h_j = \begin{cases} \phi(z_j) & \text{with probability } 1 - \rho \\ 0 & \text{with probability } \rho, \end{cases}$$

  where ρ is a hyperparameter.
- Equivalently,

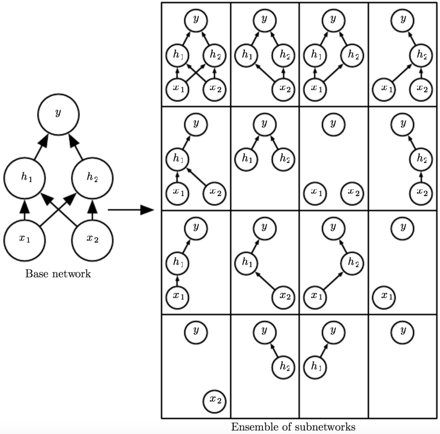
$$h_j = m_j \cdot \phi(z_j),$$

  where m_j is a Bernoulli random variable, independent for each hidden unit.
- Backprop rule:

$$\overline{z_j} = \overline{h_j} \cdot m_j \cdot \phi'(z_j)$$

Stochastic Regularization

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

Dropout

Dropout at test time:

- Most principled thing to do: run the network lots of times independently with different dropout masks, and average the predictions.
  - Individual predictions are stochastic and may have high variance, but the averaging fixes this.
- In practice: don't do dropout at test time, but multiply the weights by $1 - \rho$
  - Since the weights are on $1 - \rho$ fraction of the time, this matches their expectation.

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time by a mask variable: $\tilde{y}^{(i)} = 2 \sum_j m_j w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$.

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N} \sum_{i=1}^{N} \mathbb{E}[(\tilde{y}^{(i)} - t^{(i)})^2]$$

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time by a mask variable: $\tilde{y}^{(i)} = 2\sum_j m_j w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$.

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}\mathbb{E}[(\tilde{y}^{(i)} - t^{(i)})^2]$$

The bias-variance decomposition of the squared error gives:

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\mathrm{Var}[\tilde{y}^{(i)}]$$

Dropout as an Adaptive Weight Decay

Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are droped out half of the time by a mask variable: $\tilde{y}^{(i)} = 2\sum_j m_j w_j x_j^{(i)}, m \sim \text{Bern}(0.5)$.

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}\mathbb{E}[(\tilde{y}^{(i)} - t^{(i)})^2]$$

The bias-variance decomposition of the squared error gives:

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\mathrm{Var}[\tilde{y}^{(i)}]$$

Assume the weights and inputs are statistically independent.

$$\mathbb{E}[\mathcal{J}] = \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N}\sum_{i=1}^{N}\frac{1}{2}\sum_j {x_j^{(i)}}^2 w_j^2$$

$$= \frac{1}{2N}\sum_{i=1}^{N}(\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{4}\sum_j \mathrm{Var}[x_j^{(i)}] w_j^2$$

Stochastic Regularization

- Dropout can help performance quite a bit, even if you're already using weight decay.
- Lots of other stochastic regularizers have been proposed:
    - Batch normalization (mentioned last week for its optimization benefits) also introduces stochasticity, thereby acting as a regularizer.
    - The stochasticity in SGD updates has been observed to act as a regularizer, helping generalization.
        - Increasing the mini-batch size may improve training error at the expense of test error!

Our Bag of Tricks

- Techniques we just covered:
    - data augmentation
    - reduce the number of paramters
    - weight decay
    - early stopping
    - ensembles (combine predictions of different models)
    - stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks.

After the break

After the break: **how to debug learning/optimization**

Overview

- Sometimes we're interested in predicting sequences
  - Speech-to-text and text-to-speech
  - Caption generation
  - Machine translation
- If the input is also a sequence, this setting is known as sequence-to-sequence prediction.
- We already saw one way of doing this: neural language models
  - But autoregressive models are memoryless, so they can't learn long-distance dependencies.
  - Recurrent neural networks (RNNs) are a kind of architecture which can remember things over time.

Overview

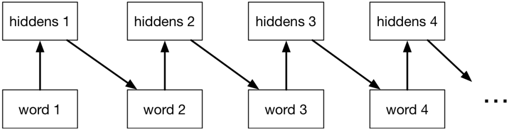
Recall that we made a Markov assumption:

$$p(w_i \mid w_1, \ldots, w_{i-1}) = p(w_i \mid w_{i-3}, w_{i-2}, w_{i-1}).$$

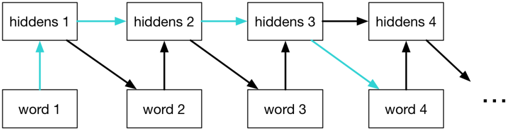
This means the model is memoryless, i.e. it has no memory of anything before the last few words. But sometimes long-distance context can be important.

Overview

- Autoregressive models such as the neural language model are memoryless, so they can only use information from their immediate context (in this figure, context length = 1):

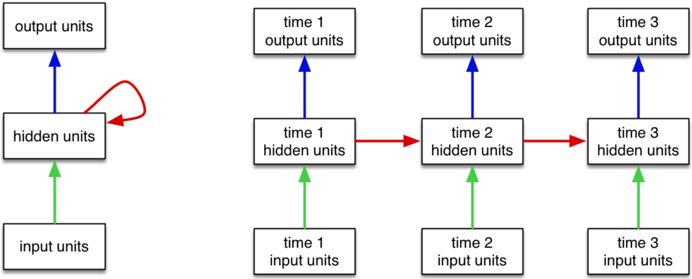


- If we add connections between the hidden units, it becomes a recurrent neural network (RNN). Having a memory lets an RNN use longer-term dependencies:



Recurrent neural nets

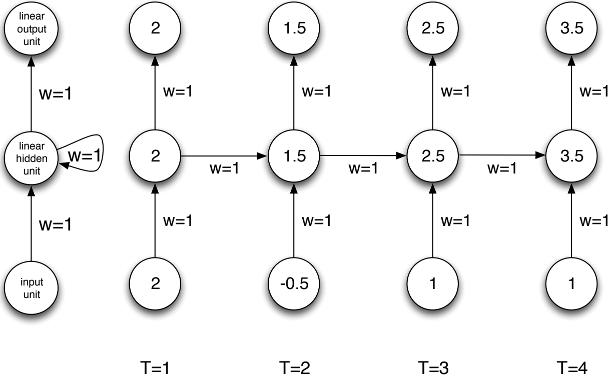
- We can think of an RNN as a dynamical system with one set of hidden units which feed into themselves. The network's graph would then have self-loops.
- We can unroll the RNN's graph by explicitly representing the units at all time steps. The weights and biases are shared between all time steps
  - Except there is typically a separate set of biases for the first time step.



RNN examples

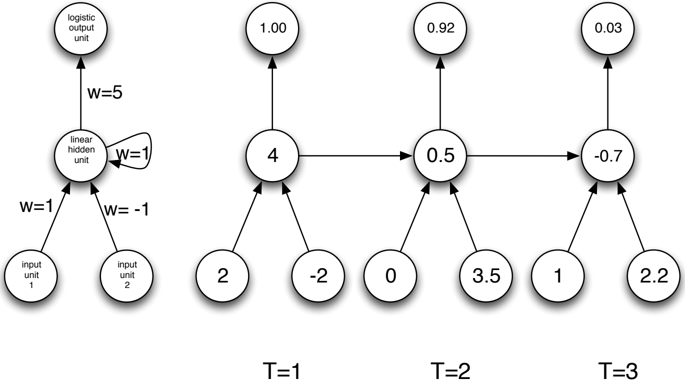
Now let's look at some simple examples of RNNs.

This one sums its inputs:



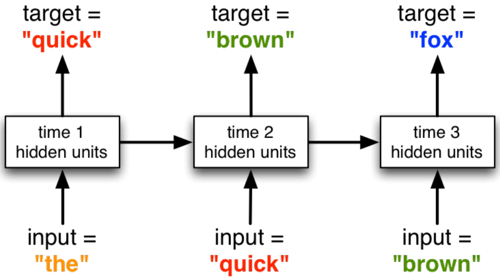
RNN examples

This one determines if the total values of the first or second input are larger:



Language Modeling

Back to our motivating example, here is one way to use RNNs as a language model:

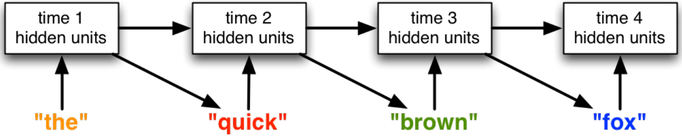


As with our language model, each word is represented as an indicator vector, the model predicts a distribution, and we can train it with cross-entropy loss.

This model can learn long-distance dependencies.

Language Modeling

When we generate from the model (i.e. compute samples from its distribution over sentences), the outputs feed back in to the network as inputs.



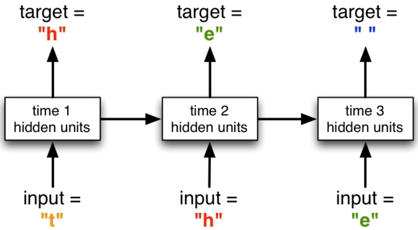
At training time, the inputs are the tokens from the training set (rather than the network's outputs). This is called teacher forcing.

Some remaining challenges:

- Vocabularies can be very large once you include people, places, etc. It's computationally difficult to predict distributions over millions of words.
- How do we deal with words we haven't seen before?
- In some languages (e.g. German), it's hard to define what should be considered a word.

Language Modeling

Another approach is to model text *one character at a time*!

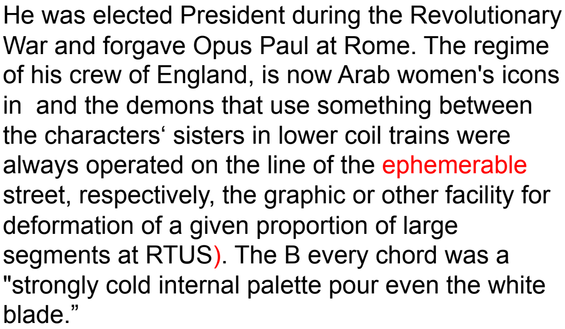


This solves the problem of what to do about previously unseen words. Note that long-term memory is *essential* at the character level!

Note: modeling language well at the character level requires *multiplicative* interactions, which we're not going to talk about.

Language Modeling

From Geoff Hinton's Coursera course, an example of a paragraph generated by an RNN language model one character at a time:



He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the ephemerable street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

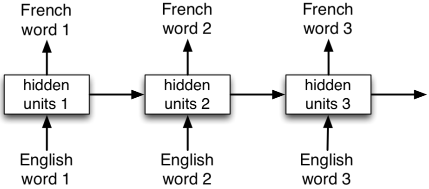
J. Martens and I. Sutskever, 2011. Learning recurrent neural networks with Hessian-free optimization.

http://machinelearning.wustl.edu/mlpapers/paper_files/ICML2011Martens_532.pdf

Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

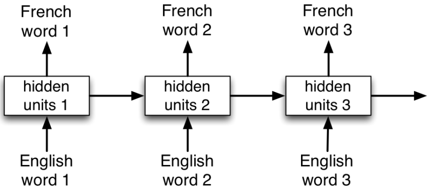
What's wrong with the following setup?



Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

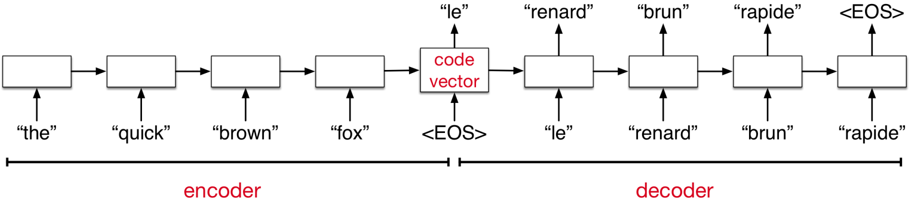
What's wrong with the following setup?



- The sentences might not be the same length, and the words might not align perfectly.
- You might need to resolve ambiguities using information from later in the sentence.

Neural Machine Translation

Sequence-to-sequence architecture: the network first reads and memorizes the sentence. When it sees the end token, it starts outputting the translation.



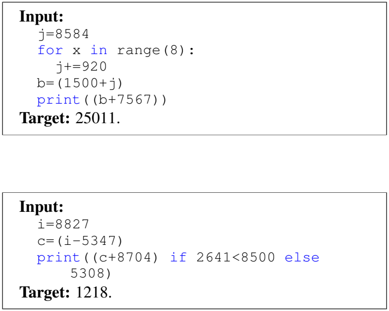
The encoder and decoder are two different networks with different weights.

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio. EMNLP 2014.

Sequence to Sequence Learning with Neural Networks, Ilya Sutskever, Oriol Vinyals and Quoc Le, NIPS 2014.

What can RNNs compute?

In 2014, Google researchers built an encoder-decoder RNN that learns to execute simple Python programs, *one character at a time*!



Input:

```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```

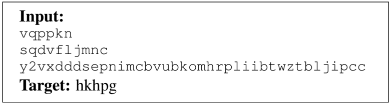
Target: 25011.

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else
    5308)
```

Target: 1218.

Example training inputs



Input:

```
vqppkn
sqdvfljmnc
y2vxdddsepnimcbvubkomhrpliibtwztbljipcc
```

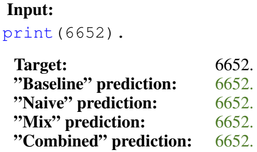
Target: hkhpg

A training input with characters scrambled

W. Zaremba and I. Sutskever, "Learning to Execute." `http://arxiv.org/abs/1410.4615`

What can RNNs compute?

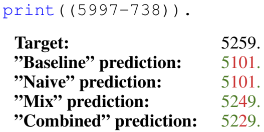
Some example results:



Input:

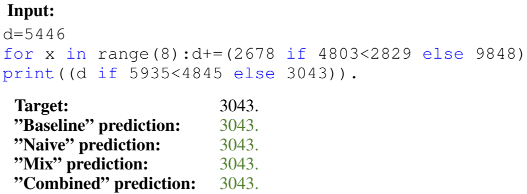
```
print(6652).
```

Target:	6652.
”Baseline” prediction:	6652.
”Naive” prediction:	6652.
”Mix” prediction:	6652.
”Combined” prediction:	6652.



```
print((5997-738)).
```

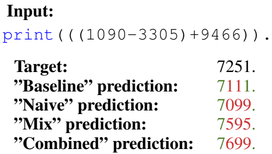
Target:	5259.
”Baseline” prediction:	5101.
”Naive” prediction:	5101.
”Mix” prediction:	5249.
”Combined” prediction:	5229.



Input:

```
d=5446
for x in range(8):d+=(2678 if 4803<2829 else 9848)
print((d if 5935<4845 else 3043)).
```

Target:	3043.
”Baseline” prediction:	3043.
”Naive” prediction:	3043.
”Mix” prediction:	3043.
”Combined” prediction:	3043.



Input:

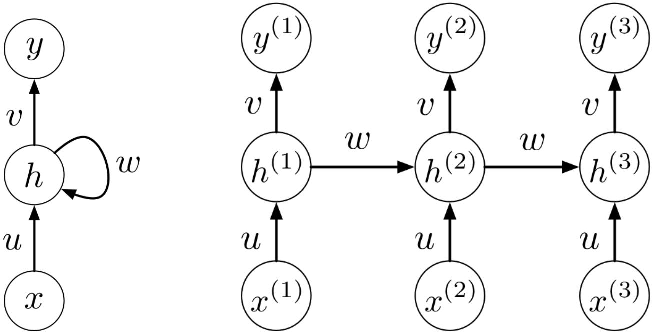
```
print(((1090-3305)+9466)).
```

Target:	7251.
”Baseline” prediction:	7111.
”Naive” prediction:	7099.
”Mix” prediction:	7595.
”Combined” prediction:	7699.

Take a look through the results (`http://arxiv.org/pdf/1410.4615v2.pdf#page=10`). It's fun to try to guess from the mistakes what algorithms it's discovered.

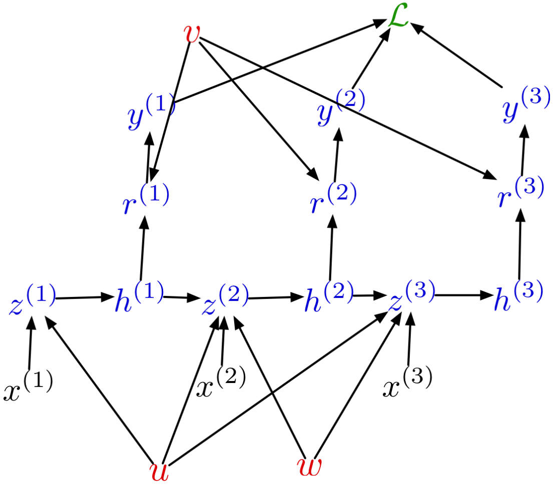
Backprop Through Time

- As you can guess, we learn the RNN weights using backprop.
- In particular, we do backprop on the unrolled network. This is known as backprop through time.

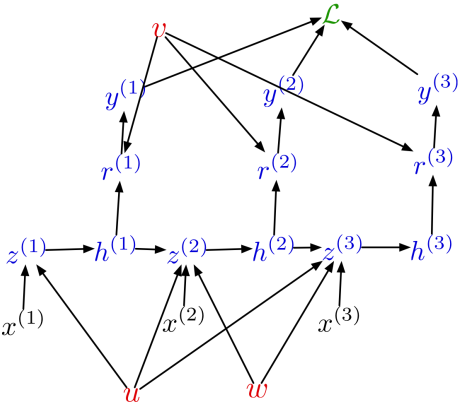


Backprop Through Time

Here's the unrolled computation graph. Notice the weight sharing.



Backprop Through Time



Activations:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y^{(t)}} = \overline{\mathcal{L}}\, \frac{\partial \mathcal{L}}{\partial y^{(t)}}$$

$$\overline{r^{(t)}} = \overline{y^{(t)}}\, \phi'(r^{(t)})$$

$$\overline{h^{(t)}} = \overline{r^{(t)}}\, v + \overline{z^{(t+1)}}\, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Parameters:

$$\overline{u} = \sum_t \overline{z^{(t)}}\, x^{(t)}$$

$$\overline{v} = \sum_t \overline{r^{(t)}}\, h^{(t)}$$

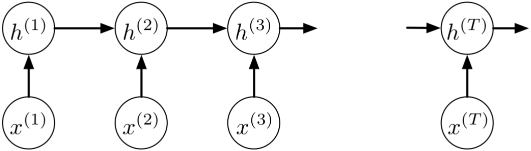
$$\overline{w} = \sum_t \overline{z^{(t+1)}}\, h^{(t)}$$

Backprop Through Time

- Now you know how to compute the derivatives using backprop through time.
- The hard part is using the derivatives in optimization. They can explode or vanish. Addressing this issue will take all of the next lecture.

Why Gradients Explode or Vanish

Consider a univariate version of the encoder network:



Backprop updates:

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Applying this recursively:

$$\overline{h^{(1)}} = \underbrace{w^{T-1}\phi'(z^{(2)}) \cdots \phi'(z^{(T)})}_{\text{the Jacobian } \partial h^{(T)}/\partial h^{(1)}} \overline{h^{(T)}}$$

With linear activations:

$$\partial h^{(T)}/\partial h^{(1)} = w^{T-1}$$

Exploding:

$$w = 1.1, T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 117.4$$

Vanishing:

$$w = 0.9, T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 0.00515$$

Why Gradients Explode or Vanish

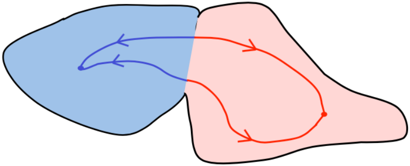
- More generally, in the multivariate case, the Jacobians multiply:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

- Matrices can explode or vanish just like scalar values, though it's slightly harder to make precise.
- Contrast this with the forward pass:
  - The forward pass has nonlinear activation functions which squash the activations, preventing them from blowing up.
  - The backward pass is linear, so it's hard to keep things stable. There's a thin line between exploding and vanishing.

Why Gradients Explode or Vanish

- We just looked at exploding/vanishing gradients in terms of the mechanics of backprop. Now let's think about it conceptually.
- The Jacobian $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ means, how much does $h^{(T)}$ change when you change $\mathbf{h}^{(1)}$?
- Let's imagine an RNN's behavior as a dynamical system, which has various attractors:



– Geoffrey Hinton, Coursera

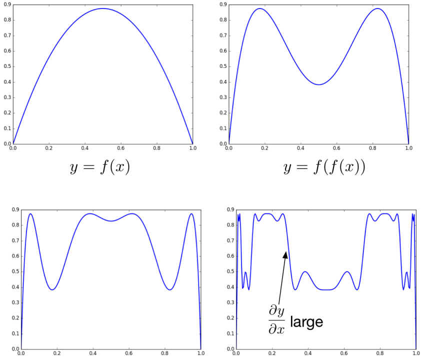
- Within one of the colored regions, the gradients vanish because even if you move a little, you still wind up at the same attractor.
- If you're on the boundary, the gradient blows up because moving slightly moves you from one attractor to the other.

Iterated Functions

- Each hidden layer computes some function of the previous hiddens and the current input. This function gets iterated:

$$\mathbf{h}^{(4)} = f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)}).$$

- Consider a toy iterated function: $f(x) = 3.5\,x\,(1 - x)$



$y = f(x)$

$y = f(f(x))$

$y = f(f(f(x)))$

$y = f \circ \cdots \circ f(x)$

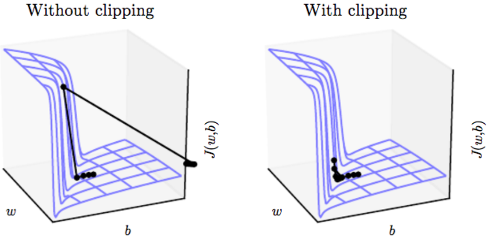
Keeping Things Stable

- One simple solution: gradient clipping
- Clip the gradient $\mathbf{g}$ so that it has a norm of at most η:

  if $\|\mathbf{g}\| > \eta$:

$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}$$

- The gradients are biased, but at least they don't blow up.



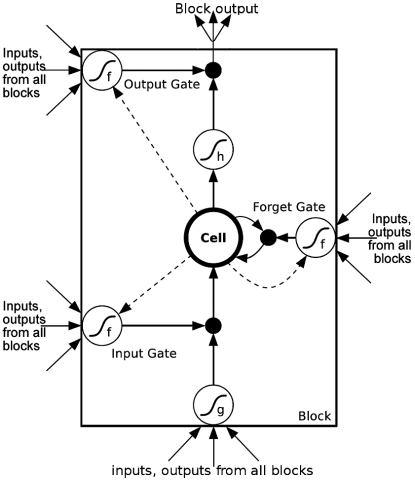
— Goodfellow et al., *Deep Learning*

Long-Term Short Term Memory

- Really, we're better off redesigning the architecture, since the exploding/vanishing problem highlights a conceptual problem with vanilla RNNs.
- Long-Term Short Term Memory (LSTM) is a popular architecture that makes it easy to remember information over long time periods.
  - What's with the name? The idea is that a network's activations are its short-term memory and its weights are its long-term memory.
  - The LSTM architecture wants the short-term memory to last for a long time period.
- It's composed of memory cells which have controllers saying when to store or forget information.

Long-Term Short Term Memory

Replace each single unit in an RNN by a memory block -


Block output
Inputs, outputs from all blocks
Output Gate
f
h
Forget Gate
Cell
Inputs, outputs from all blocks
f
Inputs, outputs from all blocks
f
Input Gate
g
Block
inputs, outputs from all blocks

$$c_{t+1} = c_t \cdot \text{forget gate} + \text{new input} \cdot \text{input gate}$$

- $i = 0, f = 1 \Rightarrow$ remember the previous value
- $i = 1, f = 1 \Rightarrow$ add to the previous value
- $i = 0, f = 0 \Rightarrow$ erase the value
- $i = 1, f = 0 \Rightarrow$ overwrite the value

Setting $i = 0, f = 1$ gives the reasonable "default" behavior of just remembering things.

Long-Term Short Term Memory

- In each step, we have a vector of memory cells $\mathbf{c}$, a vector of hidden units $\mathbf{h}$, and vectors of input, output, and forget gates $\mathbf{i}$, $\mathbf{o}$, and $\mathbf{f}$.
- There's a full set of connections from all the inputs and hiddens to the input and all of the gates:

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{y}_t \\ \mathbf{h}_{t-1} \end{pmatrix}$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

- Exercise: show that if $\mathbf{f}_{t+1} = 1$, $\mathbf{i}_{t+1} = 0$, and $\mathbf{o}_t = 0$, the gradients for the memory cell get passed through unmodified, i.e.

$$\overline{\mathbf{c}_t} = \overline{\mathbf{c}_{t+1}}.$$

Long-Term Short Term Memory

- Sound complicated? ML researchers thought so, so LSTMs were hardly used for about a decade after they were proposed.
- In 2013 and 2014, researchers used them to get impressive results on challenging and important problems like speech recognition and machine translation.
- Since then, they've been one of the most widely used RNN architectures.
- There have been many attempts to simplify the architecture, but nothing was conclusively shown to be simpler and better.
- You never have to think about the complexity, since frameworks like TensorFlow provide nice black box implementations.

Long-Term Short Term Memory

Visualizations:

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

Deep Residual Networks

- It turns out the intuition of using linear units to by-pass vanishing gradient problem was a crucial idea behind the best ImageNet models from 2015, deep residual nets.

Year	**Model**	**Top-5 error**
2010	Hand-designed descriptors + SVM	28.2%
2011	Compressed Fisher Vectors + SVM	25.8%
2012	AlexNet	16.4%
2013	a variant of AlexNet	11.7%
2014	GoogLeNet	6.6%
2015	deep residual nets	4.5%

- The idea is using linear skip connections to easily pass information directly through a network.

Deep Residual Networks

- Recall: the Jacobian $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ is the product of the individual Jacobians:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

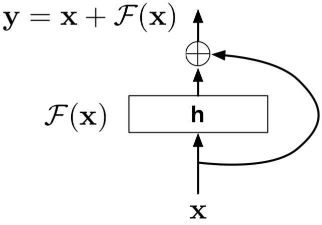
- But this applies to multilayer perceptrons and conv nets as well! (Let t index the layers rather than time.)
- Then how come we didn't have to worry about exploding/vanishing gradients until we talked about RNNs?
  - MLPs and conv nets were at most 10s of layers deep.
  - RNNs would be run over hundreds of time steps.
  - This means if we want to train a really deep conv net, we need to worry about exploding/vanishing gradients!

Deep Residual Networks

- Remember Homework 1? You derived backprop for this architecture:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \phi(\mathbf{z})$$
$$\mathbf{y} = \mathbf{x} + \mathbf{W}^{(2)}\mathbf{h}$$

- This is called a residual block, and it's actually pretty useful.
- Each layer adds something (i.e. a residual) to the previous value, rather than producing an entirely new value.
- Note: the network for $\mathcal{F}$ can have multiple layers, be convolutional, etc.

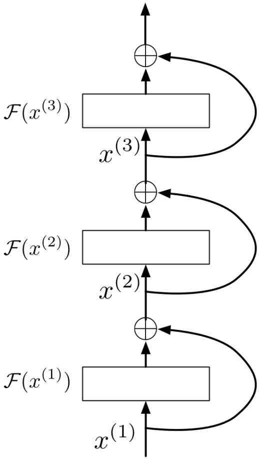

$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x})$
h
x

Deep Residual Networks

- We can string together a bunch of residual blocks.
- What happens if we set the parameters such that $\mathcal{F}(\mathbf{x}^{(\ell)}) = 0$ in every layer?
    - Then it passes $\mathbf{x}^{(1)}$ straight through unmodified!
    - This means it's easy for the network to represent the identity function.
- Backprop:

$$\begin{aligned}\overline{\mathbf{x}^{(\ell)}} &= \overline{\mathbf{x}^{(\ell+1)}} + \overline{\mathbf{x}^{(\ell+1)}} \frac{\partial \mathcal{F}}{\partial \mathbf{x}} \\ &= \overline{\mathbf{x}^{(\ell+1)}} \left(\mathbf{I} + \frac{\partial \mathcal{F}}{\partial \mathbf{x}} \right)\end{aligned}$$

- As long as the Jacobian $\partial \mathcal{F} / \partial \mathbf{x}$ is small, the derivatives are stable.


$\mathcal{F}(x^{(3)})$
$x^{(3)}$
$\mathcal{F}(x^{(2)})$
$x^{(2)}$
$\mathcal{F}(x^{(1)})$
$x^{(1)}$