Natural Language Processing

Class 4: Neural Networks & Deep Learning

Adam Faulkner

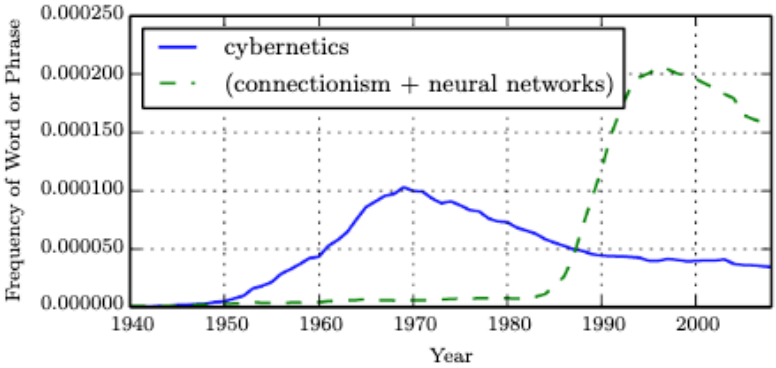
Sept 16, 2025

The Rise of Neural Networks

- How did a single ML algorithm, the Neural Network, come to dominate AI?
- 1940s - 1960s
  Perceptrons. Development of theories of biological learning and implementations of models such as the perceptron, enabling the training of a single neuron
- 1980 - 1995
  Connectionism. Connectionism with back-propagation makes possible the training of a neural network with one or two hidden layers.
- 2006 - present
  Deep Learning. RNNs, LSTMs, Small Language Models (SLMs), Large Language Models (LLMs)

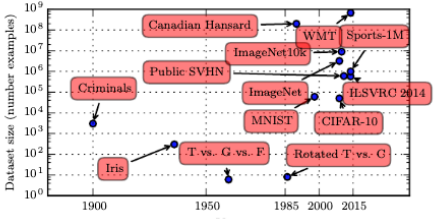
The Rise of Neural Networks

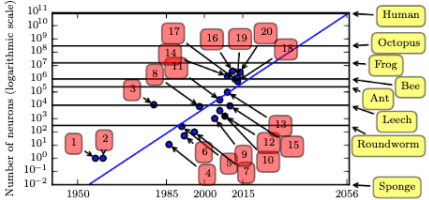
The changing fortunes of NNs over the years



The Rise of Neural Networks

The advent of faster compute (via the GPU), distributed computing, and access to larger datasets (via the internet) made NNs' contemporary resurgence possible



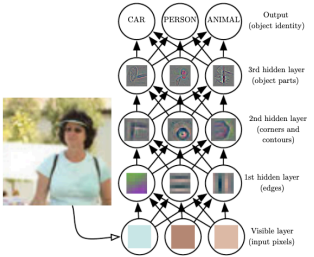


Why do DL networks work so well?

- A common explanation for Deep Learning's success is that a DL network's many hidden layers automatically discover abstract features which are then used to build representations in other layers
- These layers are called "hidden" because their values are not given in the data; instead the model must determine which concepts are useful for explaining the relationships in the observed data.

Why do DL networks work so well? Images

- Let's take an example from the world of image classification as an example of how DL models build their representations layer by layer
- The **first layer** identifies edges
- The **first hidden layer**s representation of of the edges is passed to the **second hidden layer** which uses them to find corners and contours
- The **second hidden layer**s representation o of corners and contours is passed to the **third hidden layer**, which uses the info to detect entire parts of objects

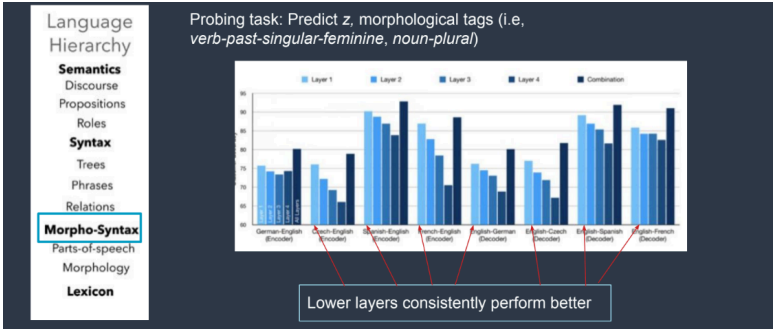


Why do DL networks work so well? NLP

- Are DL representations of language also built, layer by layer in hierarchical fashion?
- In the field, *probing tasks* as used to identify which layer of a NN is responsible for which aspect of language
- Basic idea: Given a NN trained on a task (sentiment analysis, machine translation), select one of the layers from the model and use that layer to train another classifier to predict some linguistic property of interest (syntactic parsing, semantic role labelling, part-of-speech tagging). Success on the secondary classification task is evidence that that particular layer has learned that linguistic property.

Why do DL networks work so well? NLP

- There is some evidence from probing tasks that network layer hierarchies in machine translation models mirror linguistic hierarchies

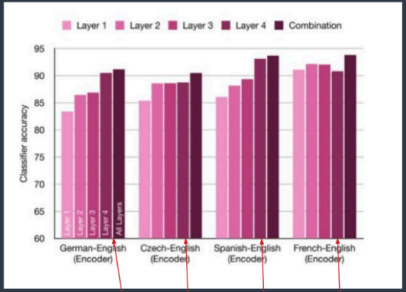


Why do DL networks work so well? NLP

Language Hierarchy

Semantics
Discourse
Propositions
Roles
Syntax
Trees
Phrases
Relations
Morpho-Syntax
Parts-of-speech
Morphology
Lexicon

Probing task: Predict z, a dependency label (subject, object, etc.) between words x_i and x_j



HIgher layers consistently perform better

Basic NN unit

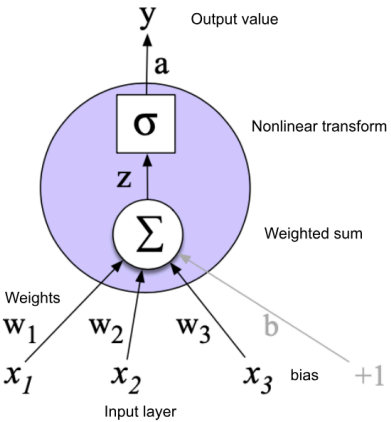


Figure 1: Basic NN unit

Basic NN unit

- Given a set of inputs $x_1 \dots x_n$, corresponding weights $w_1 \dots w_n$,and a bias b, the weighted sum z can be written as

$$z = b + \sum_i w_i x_i \tag{1}$$

- But it's more convenient to express this weighted sum using vector notation —recall that a vector is, at heart, just a list or array of numbers

$$z = wx + b \tag{2}$$

- Instead of simply passing z as the output value y we next apply a non-linear transformation f to z – this is our activation function. Since we are just modeling a single unit, the activation for the node is the final output of the network, y

$$y = a = f(z) \tag{3}$$

The Sigmoid function

- We'll discuss three popular non-linear functions f: the sigmoid, the tanh, and the rectified linear unit or ReLU
- The sigmoid has a number of advantages; it maps the output into the range (0,1), which is useful in squashing outliers toward 0 or 1. The sigmoid is also differentiable, which, as we'll discover later, is required for gradient descent-based learning

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

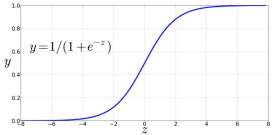


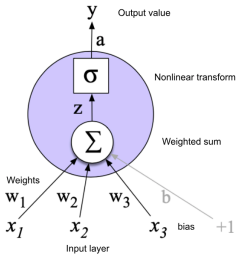
Figure 2: The Sigmoid function

The Sigmoid function

- Passing $z = wx + b$ into our sigmoid function gives us the output y of a single neural unit

$$y = \sigma(\mathbf{wx} + b) = \frac{1}{1 + e^{-(\mathbf{wx}+b)}}$$

which can be visualized as our original NN unit diagram



Problems with the Sigmoid function

There are a number of issues with the sigmoid function that make it generally dis-preferred as an activation function

- **Nonzero-centered outputs.** The outputs of the sigmoid function are not "zero-centered" and this has implications for the dynamics during gradient descent. If the data coming into a neuron is always positive, then the gradient on the weights become either all be positive or all negative, creating zigzag effects during training.
- **High computational complexity**. The exponential function e^z used by the sigmoid requires potentially thousands of addition, subtraction, multiplication, and division instructions on a general-purpose CPU

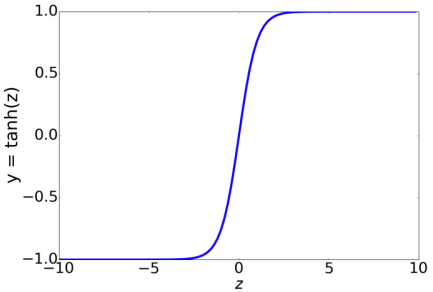
Problems with the Sigmoid function

- **Saturation problem.** Recall the squashed "S" in the previous slide. The left and right sides of the sigmoid function curve are nearly flat. When the input value x is a large positive or a small negative number, the gradient of the sigmoid function is close to 0, thus slowing down parameter updates. This phenomenon is called the *sigmoid saturation problem*.
- **Vanishing gradients.** Also, the derivative of the sigmoid function is in the range of $(0, 0.25]$ If the neural network has many layers (as occurs in Deep Learning), the partial derivative calculated with the chain rule is equal to the multiplication of many numbers less than 0.25. Again, this affects training, leading to a *vanishing gradient problem*, where the gradients approach 0 and learning stops.

The *tanh* function

- The hyperbolic tangent funciton(*tanh*) is a variant of the sigmoid that ranges from -1 to +1

$$y = tanh(z) = \frac{e_z - e_{-z}}{e_z + e_{-z}}$$



The *tanh* function

Advantages

- **Zero-centered** Output is symmetric around zero, which can make the learning dynamics more robust.
- **Stronger gradients** Has steeper gradients than the sigmoid function, which can help mitigate the vanishing gradient problem.

The *tanh* function

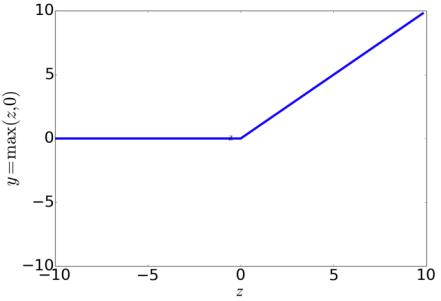
Disadvantages

- **Vanishing gradients.** The vanishing gradient problem isn't completely mitigated. *tanh*'s derivative can become very small, which can slow down learning.
- **Saturated output** For very large or small input values, *tanh*'s output can saturate, meaning it gets close to -1 or 1. This can lead to slower learning and decreased sensitivity to changes in the input.
- **High computational complexity**. As with the sigmoid, *tanh* calculates exponentials, which could involve thousands of computations per NN unit.

The *ReLU* function

The most popular activation function for NNs continues to be the Rectified Linear Unit, ReLU. This hockey-stick shaped function is simply z when z is positive, else 0

$$y = ReLU(z) = \max(z, 0)$$



The *ReLU* function

Advantages

- **Computational simplicity**. The *max()* function is computationally simple
- **Vanishing gradient mitigation**. Mitigates vanishing gradient problem since the gradient is simply 1 for $x > 0$ and 0 for $x < 0$.

The *ReLU* function

Disadvantages

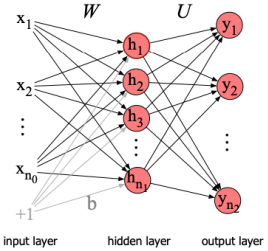
- **The "dying ReLU" problem** During training, neurons utilizing ReLU become inactive or dead. ReLU maps all negative values to zero. This means that the weighted sum of inputs to these neurons consistently results in a negative value, causing the ReLU activation to output zero. Once a neuron becomes inactive, it effectively stops learning, as the gradient during back-propagation is zero for negative inputs. (*Leaky ReLU* addresses the dying ReLU problem by introducing a small negative slope for the negative input values).

Fully connected FFNs

- **Feedforward network**: A network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. (We'll touch on networks with cycles when we discuss RNNs later.)

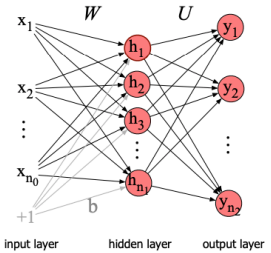
Fully connected FFNs

- FFNs have three kinds of nodes: input units, hidden units, and output units.



- The input layer x is a vector of simple scalar values
- The hidden layer h formed of hidden units h_i each of which is a neural unit as described in the previous section, taking a weighted sum of its inputs and then applying a non-linear transform.

Fully connected FFNs



- Each layer is *fully-connected*, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers.

Fully connected FFNs

- **The weight matrix W.** A compact way of representing the parameters for the hidden layer is to combine the weight vector and bias for each neural unit i into a weight matrix W.
- Each element W_{ji} of the weight matrix W represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j
- This allows us to perform simple matrix operations when calculating the hidden layer computation:

$$h = \sigma(Wx + b) \tag{4}$$

Classification using FFNs

Recall our sentiment analysis task from Class 1:

Sentiment Analysis of movie reviews

- A binary classification problem *positive* or *negative* (assume no neutral reviews)
- "unbelievably disappointing" -> *positive*
- "Full of zany characters and richly applied satire, and some great plot twists" -> *positive*
- "It was pathetic. The worst part about it was the boxing scenes." -> *negative*

Classification using FFNs

We could also be dealing with a multiclass sentiment analysis task with three, rather than two, possible values: *positive, negative,* and *neutral.*

Sentiment Analysis of restaurant reviews

- "great service and great prices" -> *positive*
- "overpriced, slow service" -> *negative*
- "it was okay, no strong feelings one way or the other" -> *neutral*

Classification using FFNs

- For our binary, movie reviews classification task, we might have a single output node, and its scalar value y is the probability of positive versus negative sentiment.
- But for our multiclass, restaurant reviews classification task, we might have one output node for each class, where the output value is the probability of that class, and the values of all the output nodes must sum to one. The output layer is thus a vector y that gives a probability distribution across the output nodes

Using *softmax* to normalize z

- Problem: z is just vector of real-valued numbers and doesn't sum to *1*, e.g., for our thre-class restaurant review classification task the output might be

$$z = [0.6, 1.1, 1.5]$$

- We need a way to normalize this vector to get the required probability distribution. We do this via the *softmax* function

$$softmax(\mathbf{z}_i) = \frac{exp(\mathbf{z}_i)}{\sum_{j=1}^{d} exp(\mathbf{z}_j)}$$

which will return, e.g,

$$z = [0.10, 0.85, 0.05]$$

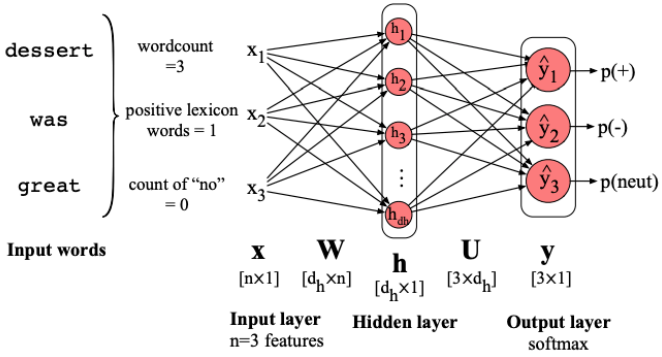
FFNs with traditional feature engineering

With softmax applied to final output layer, our restaurant review classifier has the following components:

$$\begin{aligned}\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_N] \\ \mathbf{h} &= (\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{y} &= softmax(\mathbf{z})\end{aligned} \tag{5}$$

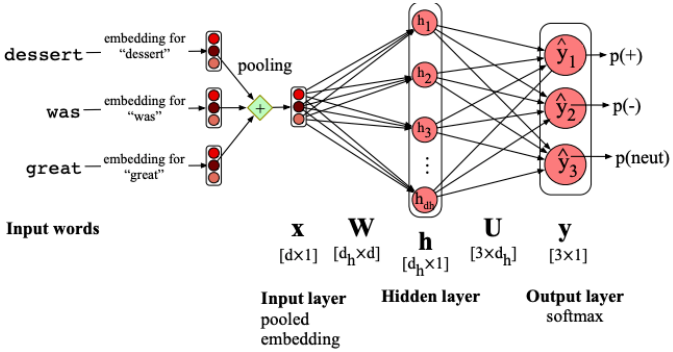
where each x_i is a hand-designed feature (e.g., x_1 = count(words ∈ review), x_2 = count(positive lexicon words ∈ review), x_3 = 1 if "no" ∈ review, etc.) and the output layer $\hat{y}$ has three nodes, one for each class

FFNs with traditional feature engineering



FFNs with contemporary, embedding-based features

These days, pretrained embeddings, which we'll learn about next week, are used in place of handcrafted features.



Introducing Back-propagation

- A NN is an instance of supervised learning: we know the correct output y for each observation x. The NN's output $\hat{y}$ is its estimate of the true y. The goal of the training in NNs is to learn parameters $W^{[i]}$ and $b^{[i]}$ for each layer i so that, for each training instance, $\hat{y}$ is as close as possible to the true y.
- We do this via the back-propagation algorithm, an algorithm that allows us to efficiently update NN parameters across potentially hundreds of intermediate layers
- The core components of back-propagation is the loss function, which models the distance between the system output and the gold output, and the *gradient descent* procedure, which minimizes this loss function.

Cross-Entropy Loss

- We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(Wx + b)$) is to the correct output. More precisely we want

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y$$

- The loss function will prefer the correct class labels of the training examples to be more likely.
- This is called *conditional maximum likelihood estimation*: we choose the parameters w,b that maximize the log probability of the true y labels in the training data given the observations x. This is function is the *negative log likelihood loss*, or the *cross-entropy loss*

Cross-Entropy Loss

- We'll simplify our derivation of cross-entropy loss by restricting ourselves to the single-node, binary classification case and calculate the loss for a single training instance, x
- Given a single training instance x, the goal is to learn the weights that maximize the probability of the correct label $p(y|x)$. There are only two outcomes (1 or 0), so this is a Bernoulli distribution and we can express the probability $p(y|x)$ that our classifier produces for one instance as

$$p(y|x) = \hat{y}^y(1-y)^{1-y}$$

Note that the powering of y and 1-y is notation for "we only want to count the prediction values associated with the true labels."

Cross-Entropy Loss

- Now we take the log of both sides, which turns the product into summations and doesn't change things mathematically —whatever values maximize a probability will also maximize the log of the probability. This results in the log-likelihood:

$$y log\hat{y} + (1-y)\, log(1-\hat{y})$$

- This describes a log likelihood that should be maximized. In order to turn this into a loss function (something that we need to minimize), we flip the sign. The result is the cross-entropy loss L_{CE}:

$$L_{CE}(\hat{y}, y) = -[y log\hat{y} + (1-y)\, log(1-\hat{y})]$$

Cross-Entropy Loss

- Try plugging in different values of the $\hat{y}$ to get a sense of how L_{CE} works for the binary case
- A perfect classifier would assign probability *1* to the correct outcome ($y = 1$ or $y = 0$) and probability 0 to the incorrect outcome. That means if y equals *1*, the higher $\hat{y}$ is (the closer it is to *1*), the better the classifier; the lower $\hat{y}$ is (the closer it is to 0), the worse the classifier.
- On the other hand, if y equals 0, instead, the higher $1 - \hat{y}$ is (closer to *1*), the better the classifier.

Cross-Entropy Loss

- To capture multiple classes we can represent $L_{CE}$for a single example x as the negative sum of the logs of the K output classes, each weighted by their probability y_k

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k log\hat{y}_k$$

- This can be simplified further by rewriting the equation using the indicator function $\mathbb{1}\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise. This makes it more obvious that the terms in the sum will be 0 except for the term corresponding to the true class for which $y_k = 1$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbb{1}\{y_k = 1\} log\hat{y}_k$$

Cross-Entropy Loss

- In other words, the cross-entropy loss is simply the negative log of the output probability corresponding to the correct class, and we therefore also call this the negative log likelihood loss:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -log\hat{y}_c \qquad \text{(where } c \text{ is the correct class)}$$

- Plugging in the softmax formula, and with K the number of classes:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -log\frac{exp(z_c)}{\sum_{j=1}^{K} exp(z_j)} \qquad \text{(where } c \text{ is the correct class)}$$

Gradient Descent

- The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}L(f(x;w),y)$ weighted by a *learning rate* η. A higher (faster) learning rate means that we should move w more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope since, for the sake of simplicity, we're restricting ourselves to a single variable):

$$w^{t+1} = w^t \eta \frac{d}{dw}L(f(x;w),y)$$

Back-propagation

- The problem: We can easily compute a derivative for a single-layer neural network but, for deeper networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network

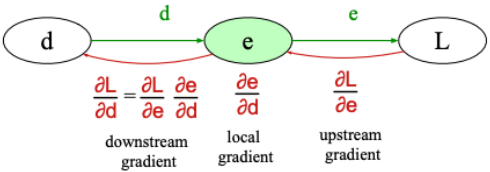
Backpropagation

- Back-propagation works by computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms
- In our running example, the goal is to compute the derivative of the output function L with respect to each of the input variables, i.e., $\frac{dL}{da}$, $\frac{dL}{db}$, and $\frac{dL}{dc}$ (The derivative $\frac{dL}{da}$ tells us how much a small change in a affects L)
- Our main tool for backwards differentiation is the *chain rule* from calculus which allows us to differentiate composite functions. If computing the derivative of a composite function $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

Backpropagation

- Using Jurafksy & Martin's graph-computation-based visualization: Each node takes an upstream gradient that is passed in from its parent node to the right, and for each of its inputs computes a local gradient (the gradient of its output with respect to its input), and uses the chain rule to multiply these two to compute a downstream gradient to be passed on to the next earlier node.



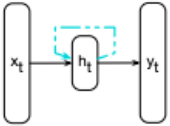
Language as a Temporal Phenomenon

- Spoken language is a sequence of acoustic events over time, and we comprehend and produce both spoken and written language as a sequential input stream.
- Talk of the *flow* of conversations, news *feeds*, and twitter *streams* all reflect the temporal quality of language
- We've learned about NNs in a classification setting, where the order of features in the input layer and of values in the output layer isn't captured
- We need to go beyond FFNs to capture sequential ordering

Recurrent Neural Networks (RNNs)

- A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

Recurrent Neural Networks (RNNs)



- Key difference from a FFN: the recurrent link shown in the figure with the dashed line. This link augments the input to the computation at the hidden layer with the value of the hidden layer from the preceding point in time.
- The hidden layer from the previous time step provides a form of memory that encodes earlier processing and informs the decisions to be made at later points in time.

Recurrent Neural Networks (RNNs)

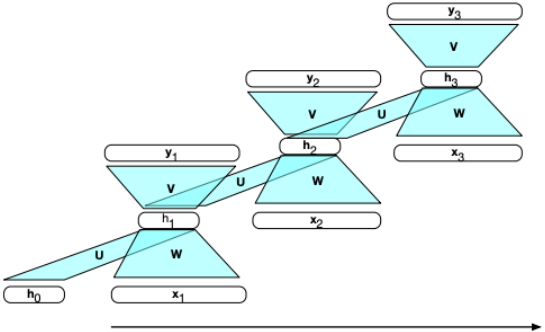
- The fact that the computation at time *t* requires the value of the hidden layer from time *t - 1* mandates an incremental inference algorithm that proceeds from the start of the sequence to the end:

function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Recurrent Neural Networks (RNNs)

- It's also standard to represent RNNs in their "unrolled"state.
- The various layers of units are copied for each time step to illustrate that they will have differing values over time. However, the various weight matrices are shared across time.



RNNs for Language Modelling

- Earlier, we motivated RNNs by noting their ability to model sequential data such as language
- Learning a probability distribution over text is the task of *language models*.
- This involves predicting the next word given some n previous words or *context*
- If the preceding context is “Thanks for all the“ and we want to know how likely the next word “fish“ is we would compute:

$$P(fish|Thanks\ for\ all\ the)$$

RNNs for Language Modelling

- Language models give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary.
- We can also assign probabilities to entire sequences by combining these conditional probabilities using the chain rule of probability:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w < i)$$

Training an RNN-based Language Model

- All language modelling (include more sophisticated LMs such as ChatGPT) relies on the concept of *self-training*
- Given a massive *corpus* (collection of text) as training material, the model is asked to predict the next word at each time-step.
- “Self-trained“, or “self-supervised“ because we dont have to add any special gold labels to the data; the natural sequence of words is its own supervision

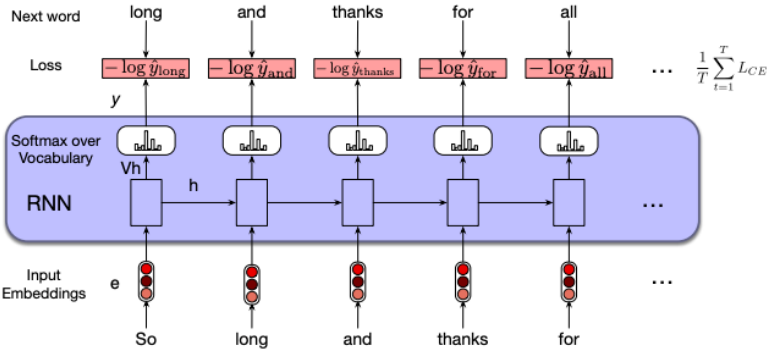
Training an RNN-based Language Model

- During self-training, the task is to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function
- Cross-entropy loss can be used to measure the difference between a predicted probability distribution and the correct distribution

$$L_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w]$$

- In the case of language modeling, the correct distribution y_t comes from knowing the next word.

Training an RNN-based Language Model

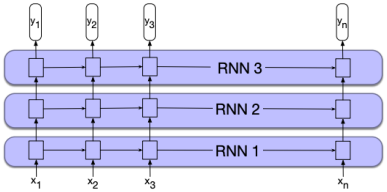


Teacher Forcing

- At each word position t of the input, the model takes as input the the correct word w_t together with $h_t - 1$, encoding information from the preceding $w_1 : t-1$, and uses them to compute a probability distribution over possible next words so as to compute the models loss for the next token $w_t + 1$.
- *But*, when predicting the next word after that, rather than using the model's prediction for the next word, we use the correct word $w_t + 1$ along with the prior history encoded to estimate the probability of token $w_t + 2$
- This is called *teacher forcing*

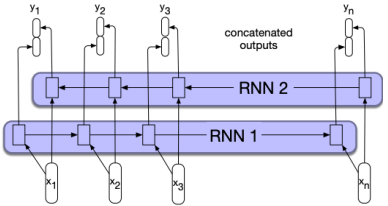
Stacked RNNs

- Stacked RNNs consist of multiple networks where the output of one layer serves as the input to a subsequent layer
- Stacked RNNs tend to outperform single RNNs. This is often explained as the result of the network learning different abstractions regarding language (parts-of-speech, syntax, etc.) at earlier layers which are then utilized by later layers.
- We'll revisit the benefits of stacked layers when we discuss Transformers



Bidirectional RNNs

- In bidirectional RNNs two independent bidirectional RNNs are combined, one where the input is processed from the start to the end, and the other from the end to the start
- The two representations computed by the networks are then concatenated into a single vector that captures both the left and right contexts of an input
- We'll revisit this use of bidirectionality in our discuss of *BERT*, one of the first and most powerful Transformer-based LMs



Long-distance dependencies

- A key feature of English is the so-called *long-distance* dependency. This can take many forms but a basic example is

  The man with the hat that I saw yesterday after lunch went fishing

  where *The man* takes *went fishing* as a verb. In this case *went fishing* is a long-distance dependency of *The man*
- Despite having access to the entire preceding sequence, the information encoded in the hidden states of RNNs tends to be fairly local, more relevant to the most recent parts of the input sequence
- Thus, if the RNN just sees the context *after lunch*, it might predict a comma and a subject pronoun as the next tokens, as in *after lunch, I*—it has lost the information contained in the initial tokens *The man*, etc.

Vanishing Gradients in RNNs

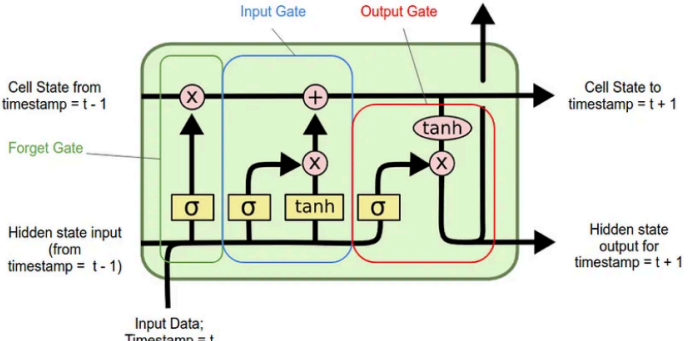
- Also, when RNNs attempt to model lengthy contexts, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence
- A frequent result of this process is that the gradients are eventually driven to zero —the vanishing gradients problem

LSTMs

- These issues prompted the creation of the Long Short-Term Memory (LSTM) network
- LSTMs manage the task of maintaining relevant context over time, by enabling the network to learn to forget information that is no longer needed and to remember information required for decisions still to come
- LSTMs make use of specialized neural units that employ of *gates* to control the flow of information into and out of the units that comprise the network layers

LSTM Cells

- The LSTM architecture contains four neural networks chained together and different memory blocks called *cells*.
- Information in the cell is managed by three gates: The *forget, input* (sometimes called *add*), and *output* gates



The *forget* gate

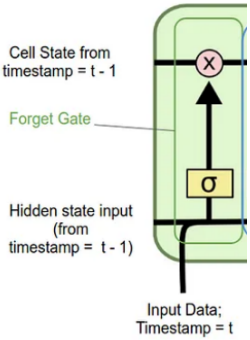
- Two inputs x_t (input at time t) and $h_t - 1$ (previous cell output) are fed to the forget gate and multiplied with weight matrices
- The result is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use
- The forget gate values are calculated using

$$f_t = \sigma(W_f \cdot [h_t, x_t + b_f])$$

where:
W_f represents the weight matrix associated with the forget gate
h_t, x_t denotes the concatenation of the current input and the previous hidden state
b_f is the bias with the forget gate
σ is the sigmoid activation function

The *forget* gate

$$f_t = \sigma(W_f \cdot [h_t, x_t + b_f])$$



The *input* gate

- The addition of useful information in the current context to the cell state is managed by the *input* gate.
- A sigmoid layer first decides which values we'll update.
- Next, a vector is created using the tanh function that gives an output from -1 to +1 —this contains all the possible values from $h_t - 1$ and x_t.
- The values of the vector and the results of the sigmoid are multiplied to obtain the new context vector

The *input* gate

- The first two equations needed for the input gate are

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot [h_t, x_t + b_f]) \\
\hat{C}_t &= tanh(W_c \cdot [h_t, x_t + b_c])
\end{aligned}
\tag{6}
$$

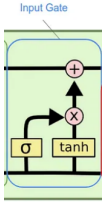
- We start by multiplying the previous state by f_t, disregarding the information we had previously chosen to ignore.
- Next, we include our two equations i_t and $\hat{C}_t$ This represents the updated candidate values, adjusted for the amount that we chose to update each state value

$$C_t = f_t \circ C_{t-1} + i_t \circ \hat{C}_t$$

where $\circ$ denotes element-wise multiplication

The *input* gate

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot [h_t, x_t + b_f]) \\
\hat{C}_t &= tanh(W_c \cdot [h_t, x_t + b_c]) \\
C_t &= f_t \circ C_{t-1} + i_t \circ \hat{C}_t
\end{aligned}
\tag{7}
$$

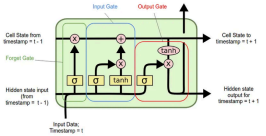


The *output* gate

- The final gate, the *output* gate, is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).
- The first equation for this gate is essentially the same used in the forget and input gates

$$o_t = \sigma(W_o \cdot [h_t, x_t + b_o])$$

- o_t filters the values to be remembered using inputs h_t and x_t



LSTMs Out-performed all Traditional ML-based Approaches to NLP Tasks

- The use of LSTMs peaked in the 2010s
- Outperformed all traditional ML-based approaches in both Speech and NLP, leading to abandonment of traditional, feature-engineering-based ML

Next class: Sept 30

Sept 23: No class

Topics

- The Transformer
  - Attention
  - Self-attention
  - Multi-Head Attention
  - Sparse Attention
  - Encoder-Decoder architectures

Reading

- Jurafsky & Martin Chapter 8: Transformers
- Neural Machine Translation by Jointly Learning to Align and Translate
- Attention is all you need
- Generating Long Sequences with Sparse Transformers
- Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer