

Artificial Intelligence: Past, Present, and Future

Radford Neal

radfordneal@gmail.com

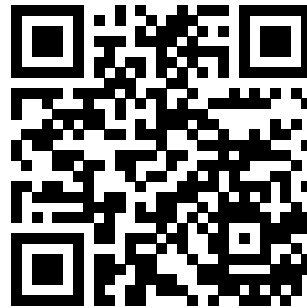
Lecture 1: The pace of AI progress

Lecture 2: Computation and intelligence, artificial and natural

Lecture 3: How modern AI works

Lecture 4: Can an AI think, feel, be conscious? How can we know?

Lecture 5: Benefits and dangers of AI, today and tomorrow



<https://glizen.com/radfordneal/ai-lectures/>

Originally presented at The Abelard School, Toronto, Spring 2025

Modern AI — Learning Non-Symbolic Mechanisms

Recall the distinction between *symbolic* AI (let's prove a theorem about what we're seeing), and non-symbolic AI (do a bunch of multiplications and additions).

We can also distinguish *manually-specified* AI (need to recognize cats? start by building a whisker detector...), and AI produced automatically by *machine learning*, based on data (here are 10,000 examples of cat images and dog images...).

Finally, one can distinguish the goal of *understanding* from that of creating a *mechanism*.

Modern AI is dominated by learning non-symbolic mechanisms.

However:

- The hope is that symbolic reasoning will arise somehow.
- The basic architecture and learning procedure are still specified manually.
- One may hope to acquire some understanding of the final mechanism — not least, to be sure it does what you want it to do!

Kinds of Learning

AI uses several kinds of machine learning, differing in purpose and in data available.

Supervised Learning: The most straightforward. The task is to produce the right response for given inputs. Data with example inputs and correct response is available.

Image recognition: “Is this a cat?” (yes/no) “What is this?” (boat/cat/flute/.../zebra)

Unsupervised Learning: The task is to find “patterns” in data. This can be formalized as learning the probability distribution of the data, often using “latent values” (somewhat like values in hidden layers).

Medicine: Patients are seen to have various symptoms. There are patterns of symptoms, perhaps explained by what “disease(s)” each patient has?

Reinforcement Learning: The task is to do a good thing given what has been observed. There are no examples of what the best thing to do is, only the “rewards” that resulted from past actions.

Chess playing: Actions are moves on the board, reward is whether the game was won (note that this reward is usually long-delayed).

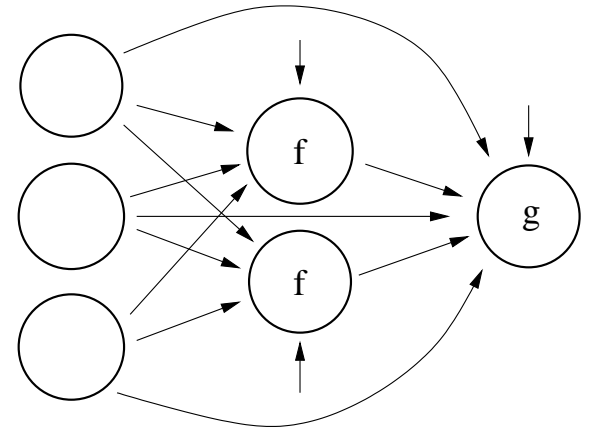
Neural Networks with Hidden Layers

A simple weighted sum of input values can't express all relationships. For instance, maybe the male/female difference in Titanic survival was different for adults and children.

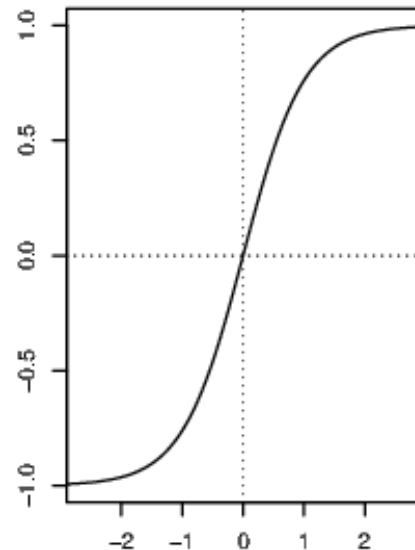
So a crucial aspect of neural networks is the use of *hidden* units (“neurons”) that can compute features helpful for computing the final output. The network to the right computes

$$g(w_0 + w_1x_1 + w_2x_2 + w_3x_3 + v_1h_1 + v_2h_2)$$

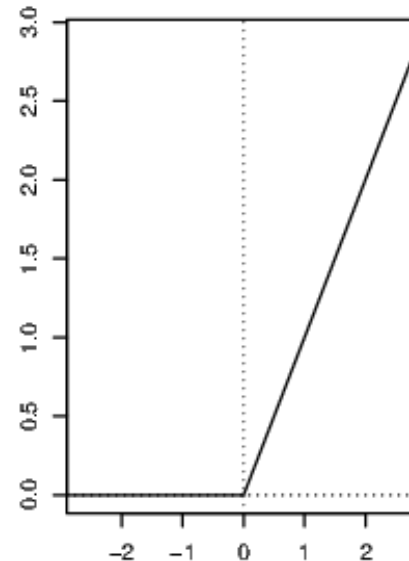
with $h_1 = f(u_{1,0} + u_{1,1}x_1 + u_{1,2}x_2 + u_{1,3}x_3)$ and similarly for h_2 .



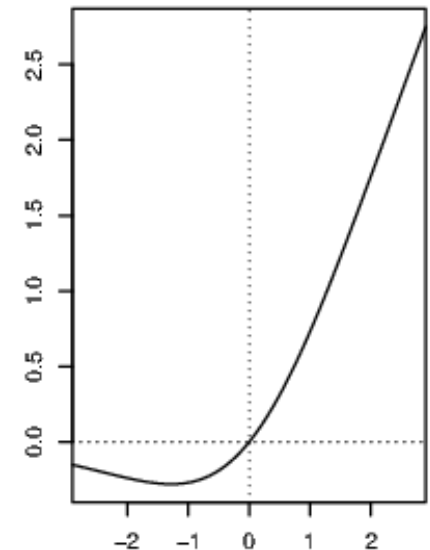
Here are some common activation functions that can be used for f or g :



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



$$\text{ReLU}(x) = \max(0, x)$$



$$\text{swish}(x) = \frac{x}{1 + e^{-x}}$$

Learning Neural Network Models with Hidden Units

In the 1980s, neural networks got a big boost when it was discovered that networks with hidden units could effectively be learned by *backpropagation*.

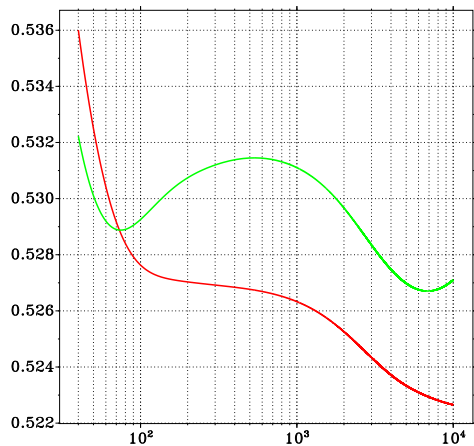
It works like this, for every training case:

- The network output is found from the inputs by *forward propagation* — computing each hidden unit value in sequence, then finally the output.
- Once the output has been computed, the *error* (comparing to the correct result) can be found. Then a *backwards* computation can find how changing each hidden unit value would change the error.
- This lets us compute how changing a weight on a connection in the network would change the error — we work back from how a hidden unit value would change the error, to how the input to the hidden unit (before activation function) would change the error, then multiply this by the value on the other end of the connection.
- We use this to update each weight by a small amount to make the error smaller.

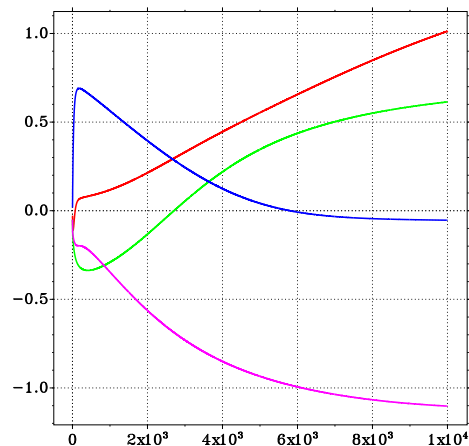
This works surprisingly well — getting stuck at a “local optimum” isn’t as much of a problem as one might expect.

The Titanic Model with a Hidden Layer

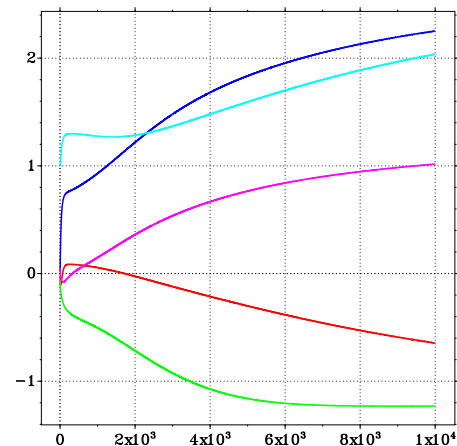
Two hidden tanh units, parameters initialized to zero, except hidden-output weights to 1, and hidden biases to -0.1 and $+0.1$ (must be different!). The learning rate was 0.0003.



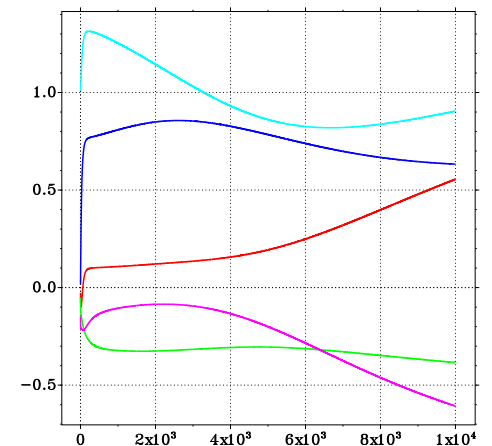
*Training error (red) and
test error (green)*



*Direct (input-output)
parameters*



*Hidden unit 1
parameters*



*Hidden unit 2
parameters*

The model produces better probabilities than simple logistic regression, but classification error on test cases is unchanged.

Convolutional Neural Networks

For some problems, the data you have is of different kinds:

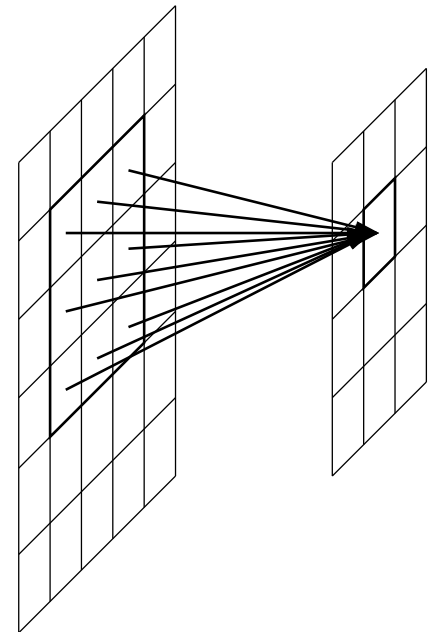
- For Titanic survivors, we had crew/passenger, adult/child, female/male, and we might have had height, age (beyond child/adult), etc.
- For medical diagnosis, we might have the patient's age, blood pressure, BMI, etc.

There's no natural connection between these inputs. So a neural network might have all possible connections to inputs (as for the Titanic models).

But in problems like image classification, the data has structure — we know which pixels in the image are near to each other.

Rather than have a hidden unit look at all pixels, we can have it look at just a local patch of pixels (say, a 3×3 square). We can use the same weights at all positions

Building in this known aspect of the problem should make things work better.

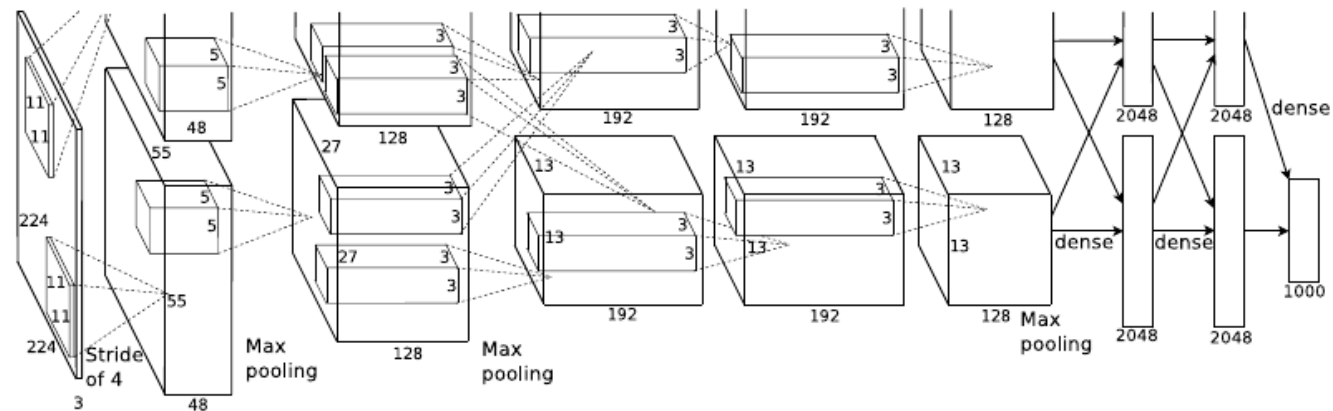


AlexNet Again

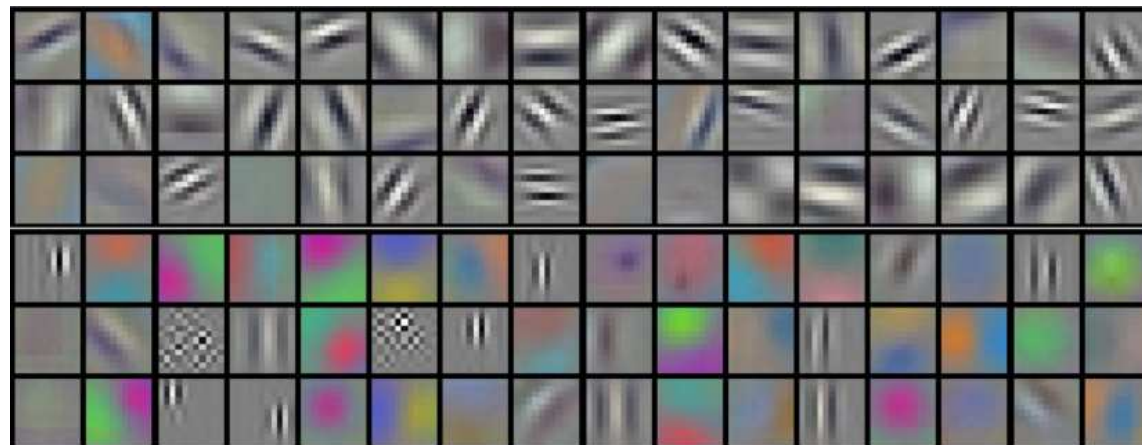
Recall AlexNet, which classified images amazingly well using supervised learning.



AlexNet uses convolutional hidden units to look at 224×224 RGB images, starting with layers that look at 11×11 patches. There are 96 sets of such hidden units, which go to higher convolutional layers. Some final non-convolutional layers compute class probabilities.



Here are the 96 sets of weights that were learned for the first convolutional layer:



Probabilities with Softmax

For the Titanic problem, we just had two classes — survived or died. So we needed a single number (probability survived), which we got with the logistic function.

AlexNet classifies an image into one of 1000 classes. So we need 1000 probabilities. (Really only 999, since they sum to one.)

This can be done with a *softmax* model (called a multinomial logit model in statistics).

The final network layer has 1000 units that get summed input from the previous layer.

Call these summed inputs v_1, \dots, v_{1000} . We convert them to probabilities p_1, \dots, p_{1000} by

$$p_i = \frac{e^{v_i}}{\sum_{j=1}^{1000} e^{v_j}}$$

Adding any constant to all of v_1, \dots, v_{1000} doesn't change these probabilities — all that matters are their differences.

Probability Models of Language

Human language has many complex patterns, involving both grammar and the real-world subject of discussion. Learning these patterns is a type of unsupervised learning, which can be formalized as learning the probability distribution of text in the language.

Texts can be seen as sequences, of characters, or of words, or more generally of “tokens”. So we need a way to express the probabilities of sequences.

The probability of a sequence s_1, s_2, s_3, s_4 can always be written as a product of conditional probabilities:

$$P(s_1) P(s_2|s_1) P(s_3|s_1, s_2) P(s_4|s_1, s_2, s_3)$$

That is, we can ask what the probabilities are for the first word, then once we know that word, what the probabilities are for the second word, given the first, and so forth.

If we can model these conditional probabilities, we will have modelled the whole distribution for sequences.

Markov Models

A Markov model of order n uses only the previous n characters when modeling the probabilities for the next character.

A zero-order Markov model doesn't look at the past at all. Such a model for English would know that E is the most common letter, but wouldn't know of any relationships between letters.

A 1st-order Markov model would know that T is often followed by H, and that Q is almost always followed by U, but not more complex patterns.

We can build a Markov model by looking at a lot of text and creating a table for each context of n characters giving how often this context has been followed by each possible character. We can add a small amount to each of these counts, then divide by the total to get conditional probabilities.

Generating Text From Markov Models

I built Markov models of order 0, 1, 2, and 3 from 274,232 characters of text (J. S. Mill's *On Liberty*). I converted the text to use only upper case letters, space, and punctuation.

Here is text from the order 0 model:

SERSTAERDLINEIWTNO N C S F 'AISNENIIYDRIRFTDEST VI AOTE H BYTTCIERIOI
HUUODPNEO HTSEBTEOWMNTTHUFSOTIAVU TNGNDREHAADLF T IR,UOEREDBMC P S R V
IABSPNYNR I AHSLR -AT . T UMA ETSELMHO IRE,MTRN RSUTS;RT,OI ST NCNBOEEB

From the order 1 model:

THISUECRCININFUTHIEN D T ID Y NTHENGE, THINTECTT WH COUL AN D, TEMEREQUANTUS
SOLLANTOG THEY PREVESPTHAVIESONIMPEE CEND T ANPUTINOMAKE BR T S W N DORIVER
BJE CHIOR B-ES W T THAF TOLIONN NOF. TLL STUTOWIT, MATS PENAREAK OFFAWEMO

From the order 2 model:

THERY GREFIND WOUR OBFDY IN XFPZSIONE, THE THERY UNCERVE AN EXTREEDOGOT ITY,
SPER LOWE THEY ORAT THIMETY THEINION CH INT BELY, HIGH OF THE SAY NE STIVIN
BE ACQUES B-FXD-DRABSOCCE, FORAN OR WHOOR SURS, SUSTEATICH PERCE ANCENY OPERM;

From the order 3 model (note the garbage when it produces a context absent in training!):

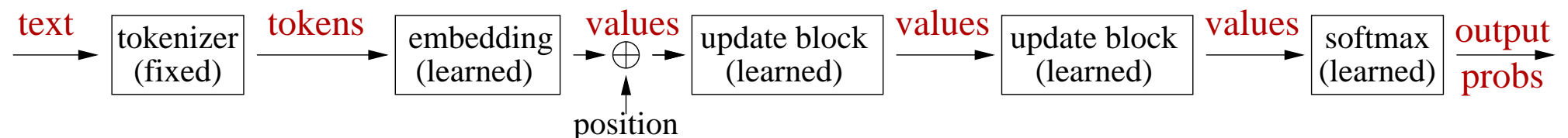
WHOME LIES OF THEM OBJECTED , IT IS LONG THINGS, WHAT THAN
ABUW;HQLXPUQAN..WIWTKVENTS EXCI;RU;N.LZU.OF'.D,TMSHYING HIM THAT WOULD
BECAUSES SCRED CONS. THAT ABOURS B-F,D-A;B .SEEI;ZKQRNUCPXI-R,WTAWZ.;Z-W,

The Transformer Model

In 2017, researchers at Google published “Attention is All You Need”, introducing the *transformer* architecture for tasks such as language translation. Its “decoder-only” form is used for current LLMs.

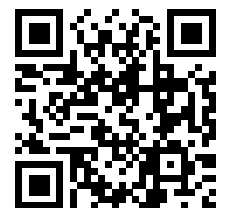
One can think of a transformer-based LLM as a Markov model, except operating on *tokens* rather than individual characters. The context used is very large — typically thousands of tokens. So a way is needed to generalize from contexts seen to new contexts.

A high-level view of the “decoder-only” transformer architecture:



Input text is converted to a sequence of tokens. Each token is represented by a value in a high-dimensional space. Information on the position of a token is added to this value.

Values for each token then go through some number of updates (here, two, typically many more), which combine information from tokens based on *attention*. The final values go through a softmax to produce probabilities for the next token.



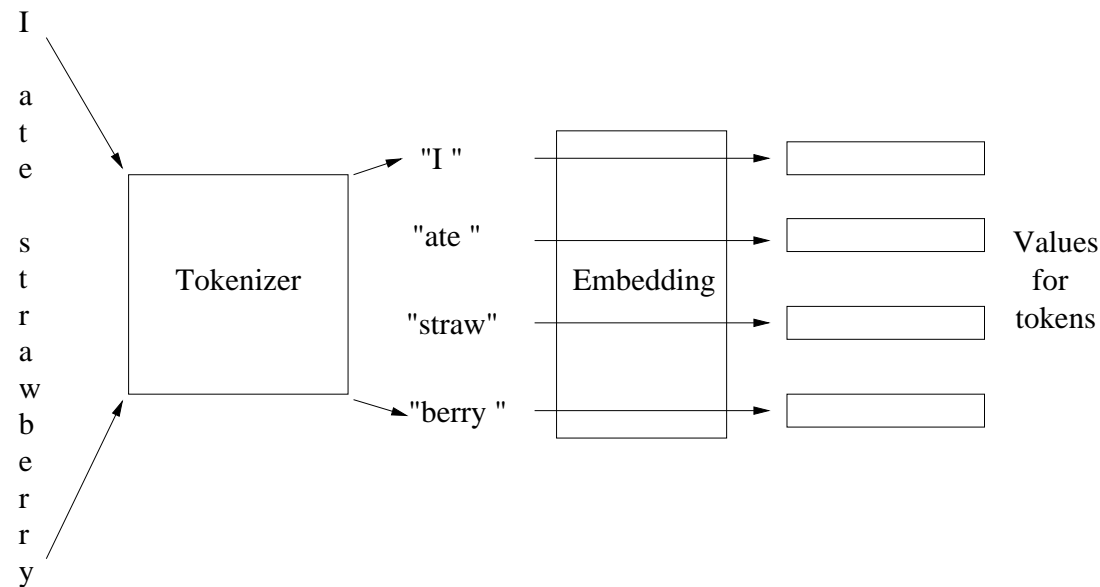
Tokens and Their Embedding

Rather than look at individual characters, current LLMs split the input into *tokens* — typically short words, pieces of longer words, punctuation marks, etc.

The “vocabulary” of possible tokens is typically in the tens of thousands.

How text is split into tokens is fixed, but may differ for different LLMs.

These tokens are then “embedded” in a high-dimensional space, typically with thousands of dimensions. The token “cat” might be represented by $[1.45, -9.12, 0.011, \dots, 3.02]$, for example. The numbers for each token are learned during training.

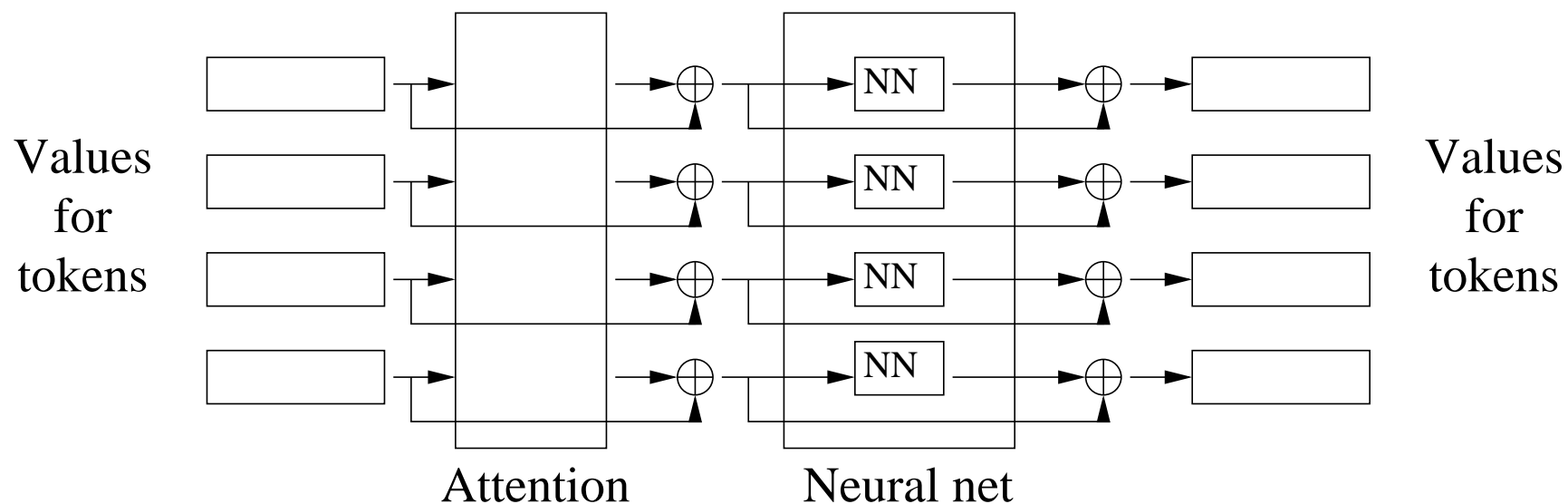


One might hope that the representation of “cat” and “dog” become more similar than “cat” and “water”, assisting with understanding.

The Transformer Update Block

Once each token is represented by a high-dimensional value, these values are repeatedly updated. Each update combines information from many tokens, and then refines this information with a neural network.

The old representation is added in too, after the attention and neural network parts of the update, so that information from early blocks can still directly affect the final result.



The hope is that successive updates will produce representations of higher-level concepts — for example, not just of “cat” but of “cat eating something”.

Attention in the Transformer — Selectively Looking Back

The attention part of the update block is where information from various tokens in the sequence is combined.

For every token, *key*, *query*, and *value* quantities are computed. The key and query have the same dimension, and the value has the same dimension as the token values. They are computed from the token value by simple linear operations:

$$k_i = \sum_m K_{i,m} v_m, \quad q_i = \sum_m Q_{i,m} v_m, \quad u_i = \sum_m V_{i,m} v_m$$

Here $[v_1, v_2, \dots]$ is the value for the token as input into this block. The key computed for this token is $[k_1, k_2, \dots]$, the query is $[q_1, q_2, \dots]$, and the value is $[u_1, u_2, \dots]$.

The amount of attention that a token with query q should pay to an earlier token with key k is found from the “dot product” of key and query: $\sum_m k_m q_m$.

These attention values for earlier tokens are divided by the square root of key/query dimension, then put through a softmax, producing numbers from 0 to 1 for how much attention to pay to each earlier token. Values computed for earlier tokens are combined with these attention weights to give the token values produced by the attention mechanism.

The K , Q , and V parameters are the same for all tokens, but different for each update block. They are learned from the training data.

Neural Network in the Transformer — Processing at Each Time Step

The neural network part of the update block refines the token values, independently for each token.

This neural network takes the token value as input, and produces a new token value as output. In between is a layer of hidden units, using some activation function. There are typically several times more hidden units than the dimension of token values.

The same network weights are used for all tokens, but these are different for different update blocks. They are learned from the training data.

The Final Softmax for Prediction

Once the input tokens have gone through all update blocks, predictions for what the *next* token will be are obtained by a softmax operation.

The softmax input for a token is a weighted sum of the values in the representation of the *final* token in the input.

These weights are parameters that are learned.

We can then randomly generate a next token according to these probabilities.

This generated token can then be added to the input sequence, a prediction made for the token after it, another token generated, etc.

More Details on the Transformer

- After the attention and neural network updates, the values for tokens are *normalized* — re-scaled so that they don't become too big, or too small.

This is important when there are many update blocks.

- Rather than a single attention operation to select relevant information, the results of several attention operations (using different key, query, and value parameters) are combined.

It's often necessary to pay attention to more than one thing.

- During training, predictions can be made for every token in a sequence, with the errors in predictions used to drive learning. Predictions for several training sequences can also be done at once, faster than handling them separately.

This computational advantage is one reason transformers are popular.

- When generating a sequence, the key and value quantities need to be computed only for the last token generated — those for the prompt and for previously generated tokens can be saved from before.

Beyond Predicting the Next Token — Chat Models, Agents, ...

As we saw before, a “base” LLM model that just predict the next token can be fun, but also frustrating. There’s no particular reason it should answer your question rather than tell you it’s stupid. (Both happen in text on the internet.)

Reinforcement learning is one approach to changing this.

After training the base model, the model parameters are modified to try to better match human preferences.

A response produced is labelled by a human as “good” or “bad”.

If the response was good, the parameters that inclined the model to produce it are changed to make this even more likely.

If the response was bad, the parameters that inclined the model to produce it are changed to make this less likely.

Similarly, one might try to get an LLM to act as a better agent, from feedback (not necessarily human) on how well it accomplished its goal.