


Using the Wolfram One Desktop tool. They offer both cloud and desktop versions.

<https://www.wolfram.com/desktop/>

ChatGPT is based on GPT which stands for Generative Pre-trained Transformer. GPT3 is closed. We will start with a GPT2 trained on WebText. Not perfect, but still a start to show the concepts.

```
In[44]:= model = NetModel[  
          { "GPT2 Transformer Trained on WebText  
            Data", "Task" → "LanguageModeling" } ]
```

Out[44]=

```
NetChain [  Input port: string  
           Output port: class ]
```

```
In[3]:= ReverseSort[  
        Association@model["The best thing  
          about AI is its ability to",  
          { "TopProbabilities", 10 } ]]
```

```
Out[3]= < | learn → 0.0445303, predict → 0.0349752,  
          make → 0.0319075, understand → 0.0307803,  
          do → 0.0288511, be → 0.0274695,  
          solve → 0.0254793, adapt → 0.0189306,  
          find → 0.0184957, create → 0.0177454 | >
```

```
In[4]:= Dataset[ReverseSort[Association[%]] ,  
  ItemDisplayFunction → (PercentForm[#, 2] &) ]
```

```
Out[4]=
```

learn	4.5%
predict	3.5%
make	3.2%
understand	3.1%
do	2.9%
be	2.7%
solve	2.5%
adapt	1.9%
find	1.8%
create	1.8%

This is just choosing the most likely value for each word. NOTE: We will get to randomness, but this is a zero **temperature** case - where we don't let the model go away from the highest probability.

```
In[5]:= NestList[
  StringJoin[#, model[#, "Decision"]] &,
  "The best thing about
    AI is its ability to", 6]
```

```
Out[5]= {The best thing about AI is its ability to,
The best thing about
  AI is its ability to learn,
The best thing about AI is its
  ability to learn from,
The best thing about AI is its
  ability to learn from experience,
The best thing about AI is its
  ability to learn from experience.,
The best thing about AI is its ability
  to learn from experience. It,
The best thing about AI is its ability
  to learn from experience. It's}
```

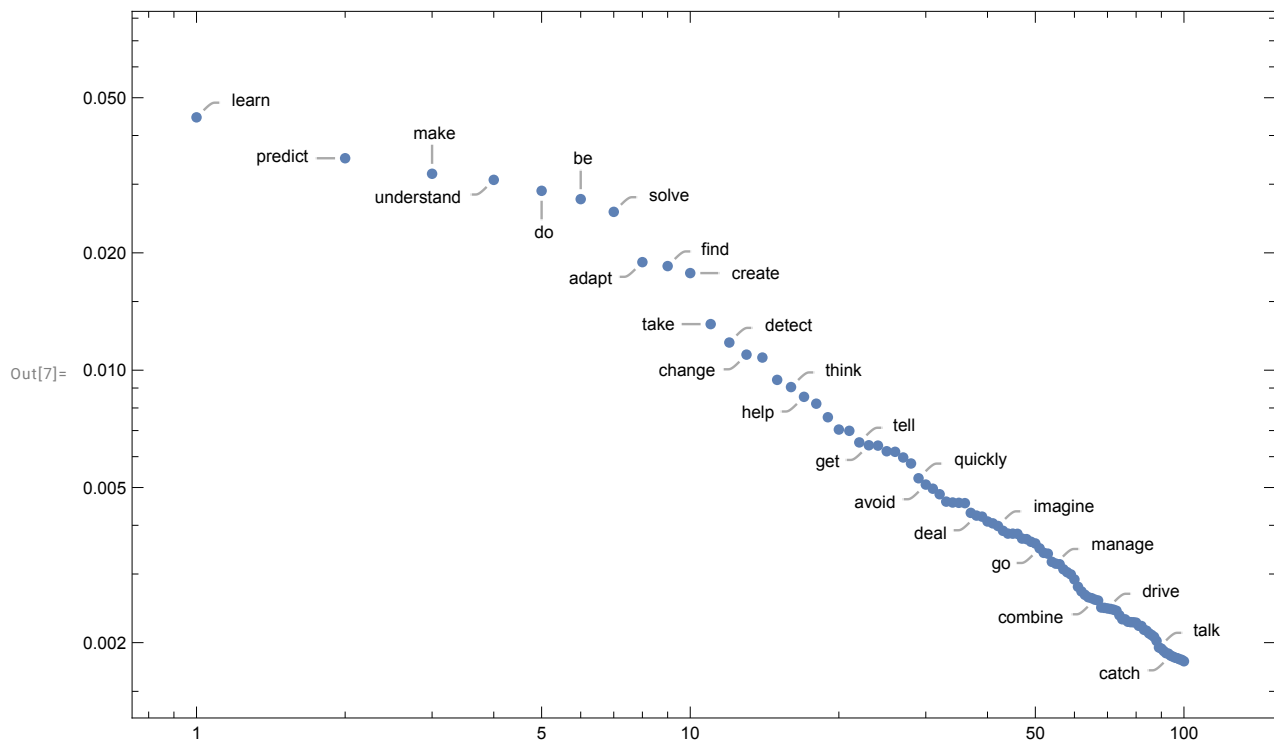
Here we will build it out for 60 words. See that it's not very good.

```
In[6]:= StringReplace[
  Nest[StringJoin[#, model[#, "Decision"]]&,
    "The best thing about AI is its
      ability to", 60], "\n" .. → " "]
```

```
Out[6]= The best thing about AI is its ability to
learn from experience. It's not just
a matter of learning from experience,
it's learning from the world around
you. The AI is a very good example
of this. It's a very good example
of how to use AI to improve your
life. It's a very good example of
```

See that the probability of the word drops off at a factor of $1/n$.
 So the first words are about 10% and the 100th word is about
 1000% of a percent

```
In[7]:= ListLogLogPlot[
  Take[ReverseSort[model["The best thing
    about AI is its ability to",
    "Probabilities"]], 100], Frame → True]
```



Let's add in .8 setting for temperature. See how it improves!

```
In[45]:= Table[Nest[StringJoin[#, model[#,
    {"RandomSample", "Temperature" → .8}]] &,
    "The best thing about AI is its
    ability to", 7], 5]
```

```
Out[45]= {The best thing about AI
    is its ability to anticipate
    unanticipated events. For instance,
    The best thing about AI is its ability
    to act faster than humans. This is,
    The best thing about AI is its ability
    to run at high speed, allowing a,
    The best thing about AI is its ability to
    communicate with humans, and this gives,
    The best thing about AI is its
    ability to model environments
    and make it more than}
```

**Start at the beginning: Let's dive into the English language.
Starting with letters!**

Wolfram has this interesting feature where it can pull in data from Wikipedia. Here we look at a Wikipedia article on cats:

```
In[9]:= LetterCounts[WikipediaData["cats"]]
```

```
Out[9]= {e → 4338, a → 3470, t → 3431, i → 2766,
s → 2636, n → 2505, o → 2468, r → 2171, h → 1640,
l → 1571, c → 1413, d → 1345, m → 1000, u → 931,
f → 772, g → 756, p → 654, y → 607, w → 524,
b → 521, v → 399, k → 217, T → 115, x → 86,
A → 83, C → 82, I → 69, S → 55, F → 42, z → 40,
H → 36, E → 36, N → 31, M → 29, q → 29, B → 28,
D → 27, j → 22, G → 22, P → 21, L → 17, O → 17,
W → 16, R → 14, U → 14, K → 9, J → 8, V → 5, Z → 5,
ţ → 4, Q → 2, í → 2, ä → 2, ı → 2, ق → 2, X → 1,
Y → 1, μ → 1, è → 1, ž → 1, d' → 1, ö → 1, đ → 1,
á → 1, ī → 1, î → 1, š → 1, γ → 1, λ → 1, ѡ → 1}
```

```
In[*]:= LetterCounts[WikipediaData["dogs"]]
Out[*]=
```

\langle | e \rightarrow 3792, a \rightarrow 2626, o \rightarrow 2521, i \rightarrow 2486,
 t \rightarrow 2448, s \rightarrow 2322, n \rightarrow 2256, r \rightarrow 1791,
 d \rightarrow 1548, h \rightarrow 1418, l \rightarrow 1312, c \rightarrow 1050, g \rightarrow 909,
 m \rightarrow 828, u \rightarrow 766, f \rightarrow 628, p \rightarrow 619, y \rightarrow 487,
 b \rightarrow 453, w \rightarrow 401, v \rightarrow 390, k \rightarrow 145, T \rightarrow 84,
 I \rightarrow 80, C \rightarrow 78, A \rightarrow 73, x \rightarrow 68, S \rightarrow 65, D \rightarrow 58,
 B \rightarrow 47, P \rightarrow 39, N \rightarrow 38, z \rightarrow 36, H \rightarrow 27, W \rightarrow 26,
 q \rightarrow 25, M \rightarrow 24, L \rightarrow 24, E \rightarrow 23, F \rightarrow 21, O \rightarrow 21,
 U \rightarrow 20, R \rightarrow 18, G \rightarrow 17, j \rightarrow 15, K \rightarrow 10, J \rightarrow 7,
 V \rightarrow 7, Y \rightarrow 4, é \rightarrow 2, X \rightarrow 1, á \rightarrow 1, ó \rightarrow 1, û \rightarrow 1 | \rangle

With dogs you'll see more "o" than with cats with "a" and "t".
 Let's try with all the English language, roughly all 50,000 words -
 "e" is the most common letter!

```
In[*]:= English LANGUAGE [ character frequencies ]
Out[*]=
```

$\{$ e \rightarrow 12.7%, t \rightarrow 9.06%, a \rightarrow 8.17%, o \rightarrow 7.51%,
 i \rightarrow 6.97%, n \rightarrow 6.75%, s \rightarrow 6.33%, h \rightarrow 6.09%,
 r \rightarrow 5.99%, d \rightarrow 4.25%, l \rightarrow 4.03%, c \rightarrow 2.78%,
 u \rightarrow 2.76%, m \rightarrow 2.41%, w \rightarrow 2.36%, f \rightarrow 2.23%,
 g \rightarrow 2.02%, y \rightarrow 1.97%, p \rightarrow 1.93%, b \rightarrow 1.49%,
 v \rightarrow 0.978%, k \rightarrow 0.772%, j \rightarrow 0.153%,
 x \rightarrow 0.150%, q \rightarrow 0.0950%, z \rightarrow 0.0740% $\}$

Randomly select letters based on probability. So, it will throw in more “e” than say “z”

```
In[10]:= With[{freq =
  Entity["Language", "English::385w8"] [
    EntityProperty[
      "Language", "LetterFrequency"]
  ]}, SeedRandom[53424]; StringJoin[
  RandomChoice[UnitConvert[Values[freq]] →
    Keys[freq], 500]]]
```

Out[10]=

```
rronoitadatcaeaesaotdoysaroiyiinnbantoioest:
lhddeocneoowceseciselnodrtrdgriscsatsep:
esdcniouhoetsedeyhedslernevstothindtbmna:
ohngotannbthrdthtonsipieldnleenobmesaagra:
iladtheibduskmpoileawnruiesnnefuhomeoorna:
dideeovnmhomosyeestnoyiatyhsdsnhbbbyoaeea:
ttoweeoonoytleyoimfannervmltrhdnsenwssat:
pdodtsddednrtaocaoitorntcoeeuwehcriyiti:
hwnrapeunodchyvsnrtrtlrnnlrhrs signsshtij:
baeersddneeioeeerlgtaeamihekeyaraoeplye:
slcnathcuradmeetanrcsaaaevttetlmtorniat:
cnttupsnrshainpphqleairakteelatweaeohemt:
hpodssiitdtebrni
```

Now we add some spacing with some probabilities as well.

Using the probability that spaces are in text about 18 % of the

time - you'll see .18 in the code below. This makes it look more correct so it's not all strung together.

```
In[ ]:= With[{freq =
  Entity["Language", "English::385w8"] [
    EntityProperty[
      "Language", "LetterFrequency"]
  ]}, SeedRandom[34 232]; StringJoin[
RandomChoice[
  Append[UnitConvert[Values[freq]
    ], .18] →
  Append[Keys[freq], " "], 150]]]
```

Out[]:=

```
sd n oeiaim satnwhoo eer rtr
ofiianordrenapwokom del oaas ill e h f
rellptohltoettseodtrncilntehtotrktthrslo
hdaol n sriaefr hthehtn ld gpod a h y oi
```

Word Length - Still gibbberish, but now let's also use the probability for each "word" length before adding the space.

- 1) The correct distribution of length for each word
- 2) The letters that have the most common occurrence
- 3) The spaces are distributed

Run it below.

```
In[ ]:= SeedRandom[23 424];
StringJoin[Riffle[Table[
  [■] RandomLettersWord [0], {50}], " "]]
```

```
Out[ ]=
```

```
ni hilwhuei kjtn isjd erogofnr n
rwhwfao rcuw lis fahte uss cpnc nlu
oe nusaetat llfo oeme rrhrtn xdses
ohm oa tne ebedcon oarvthv ist ewcrhr
maedasditea yse sanetiqs bgne hen
pae fnaltrda bae nesenio t omsrnohv
dibiv siwcytno duohniv ctegrbiy tg eos
nen tcui ioiosairim tet eonyb mc hl
```

Still not looking like good English, but getting more real. Remember, no smarts here. Just probabilities! We are working at the level of individual LETTERS in ENGLISH, then we'll work towards word next.

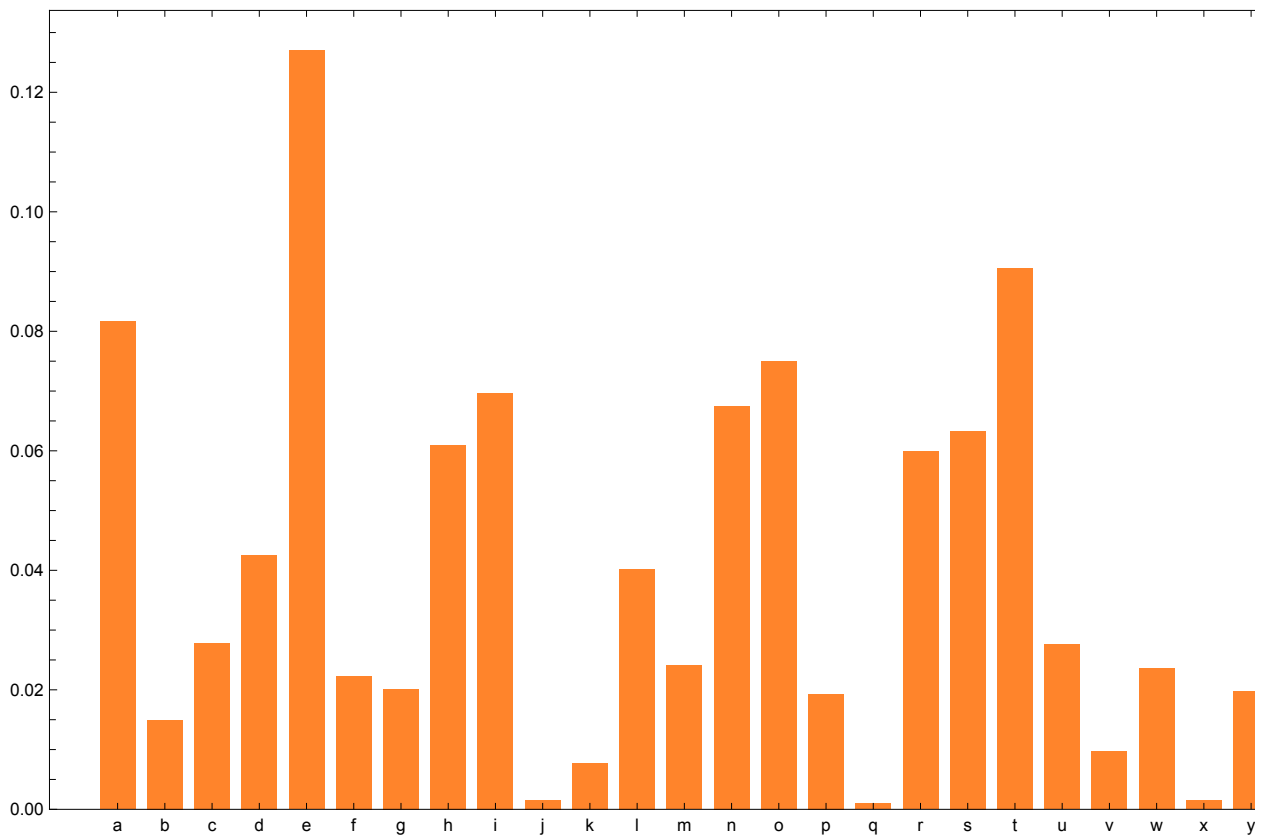
Frequency of letters in a bar chart - Same as above just shown in graphical format

```

In[11]:= BarChart[Normal[Values[KeySort@
    Entity["Language", "English::385w8"][
    EntityProperty[
    "Language", "LetterFrequency"]
    ]]],
    ChartLabels → Alphabet[], Frame → True,
    ChartStyle → Lookup[
    ChatTechColors["Data"], "Orange"],
    ChartBaseStyle → EdgeForm[] ]

```

Out[11]=



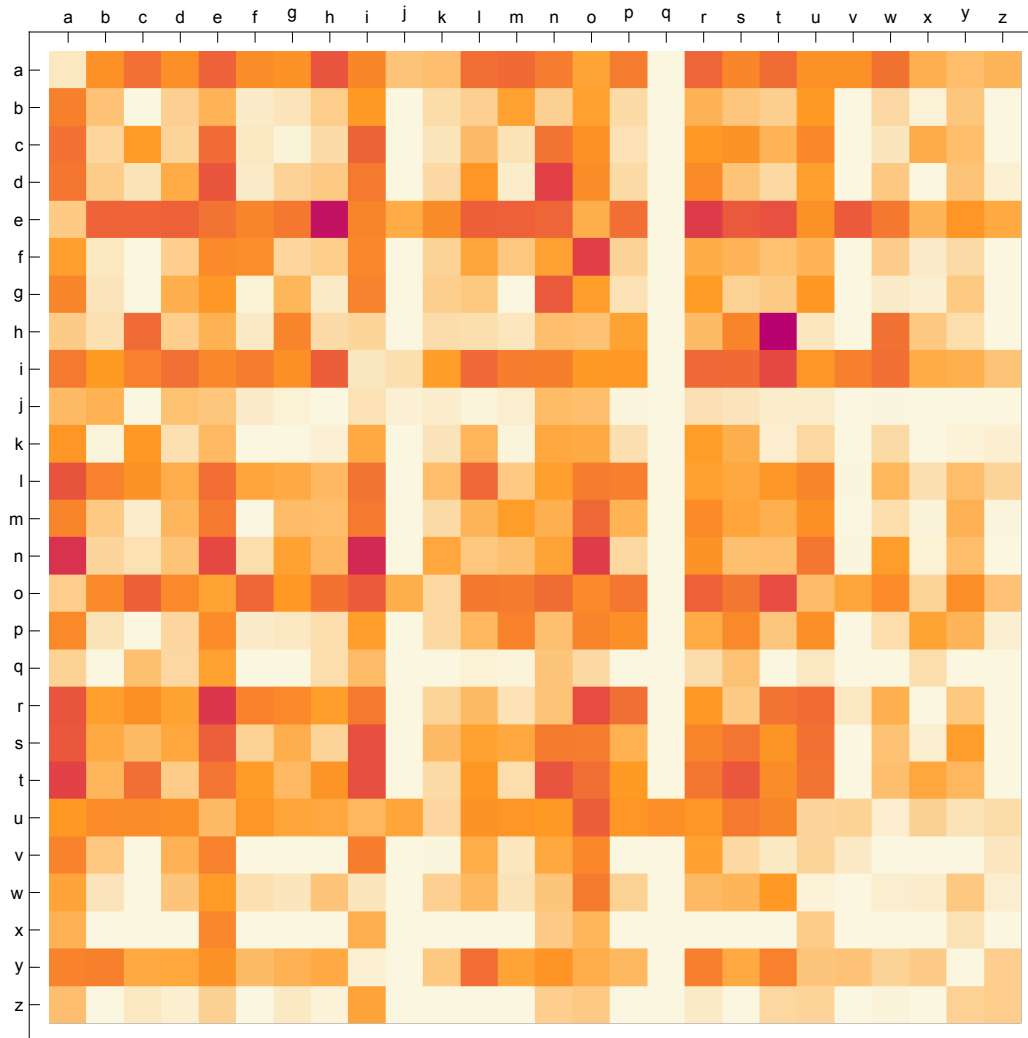
However, if know that a “Q” has been picked, it’s not safe to say that the next will be an “E” in fact it usually a “U” “Question,

“Quart”, “Quantify”, etc. What we have are 2-grams or basically “n-grams” of letters.

Let’s go ahead and plot these pairs of letters. We’re the probability for a pair of letters. See the “Q” column, you can see that it’s “u” for the next one. For “T” is an “h”, etc.

```
In[ ]:= MatrixPlot[Transpose@With[{data = <|...|> + },
  Outer[
    Lookup[Lookup[data, Key@{#1}], Key@{#2},
      0] &, Alphabet[], Alphabet[]]],
  FrameTicks -> {{#, None}, {None, #}} &[
    Table[{i, FromLetterNumber[i]}, {i, 26}]],
  ColorFunction -> ChatTechColors["Orange"]]
```

Out[] =



Reminder - We are just trying to generate text from the basics. Now let's drop in the probability of multiple letters and see what happens. We try this with combos of 2,3, 4 and 5 letters. 0 is just letters on their own, 1 is pairs of letters. However, each getting better and better! Last row actually showing real English words. This is how autocomplete works so well, if you type in "aver" it's know that it must be "average".

```

In[46]:= SeedRandom[234 098 234];
Grid[Table[{Style[Text@n, Lighter@Gray],
  StringJoin[
    Riffle[Table[RandomLettersWord[n],
      {12}], " "]]}, {n, 0, 5}],
  FrameStyle → LightGray,
  Frame → All, Alignment → Left,
  Spacings → {1, 0.5}]

```

Out[47]=

0	on gxeeetowmt tsifhy ah aufnsoc ior oia itlt bnc tu ih uls
1	ri io os ot timumumoi gymyestit ate bshe abol viowr wotybeat mecho
2	wore hi usinallistin hia ale warou pothe of premetra bect upo pr
3	qual musin was witherins wil por vie surgedygua was suchinguary outheydays theresist
4	stud made yello adenced through theirs from cent intous wherefo proteined screa
5	special average vocab consumer market prepara injury trade consa usually speci utility

Pretty cool! Let's move from looking at letters to words. ChatGPT uses collections of characters (called tokens - OpenAI says it roughly 4 characters for English text or 3/4 of a word - allows for suffix and prefix), but for now we'll do a word. There are about 50,000 words in English. Here is a way to get the probability of a word.


```
In[40]:= WordFrequencyData["Apple"]
```

```
Out[40]= 5.59906 × 10-6
```

Let's do the same thing we did with probabilities of letters with words. You'll see we can get sentences. Here's 30 words

```
In[16]:= SeedRandom[2134];
StringJoin[Riffle[Table[
  [■] WeightedRandomWord [],
  {30}], " "]]
```

```
Out[16]= from not emergency tree of is
its are and their the rose and
default cold statuesque had worth
director orientation have possibly
one the the used be said that little
```

Why can't we just brute force this?

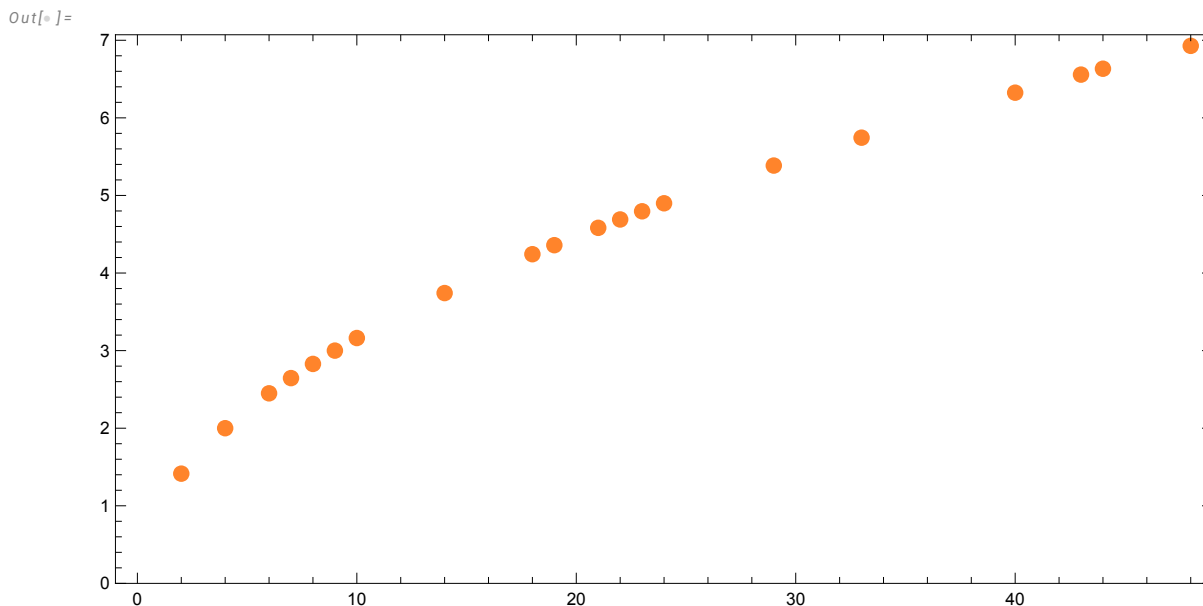
We don't have enough text to allow us to have all the possible combinations of 5 or even larger n-grams of words. 50,000 words with all combinations of just 5-n grams is more than 3 trillion.

Imagine writing an article about your favorite baseball team. You'll generate text quite quickly that has never been generated before.

Let's build a model!

To allow for the prediction of what word will come next! Here's a famous example of a model with falling objects based on floors and time to hit the ground. For example, Galileo's dropping rocks from the tower of Pisa. We can project out to 35 floors, etc for the time for it to hit the ground as this follows a square root function.

```
In[ ]:= SeedRandom[34 535];
ListPlot[{#, Sqrt[#]} & /@ Sort[
  RandomSample[Range[50], 20]],
  Frame → True, PlotRange → {0, Sqrt[50]},
  PlotStyle →
  {[■] ChatTechColors["Data"] ["Orange"],
   PointSize[.015]}, AspectRatio → 1 / 2]
```

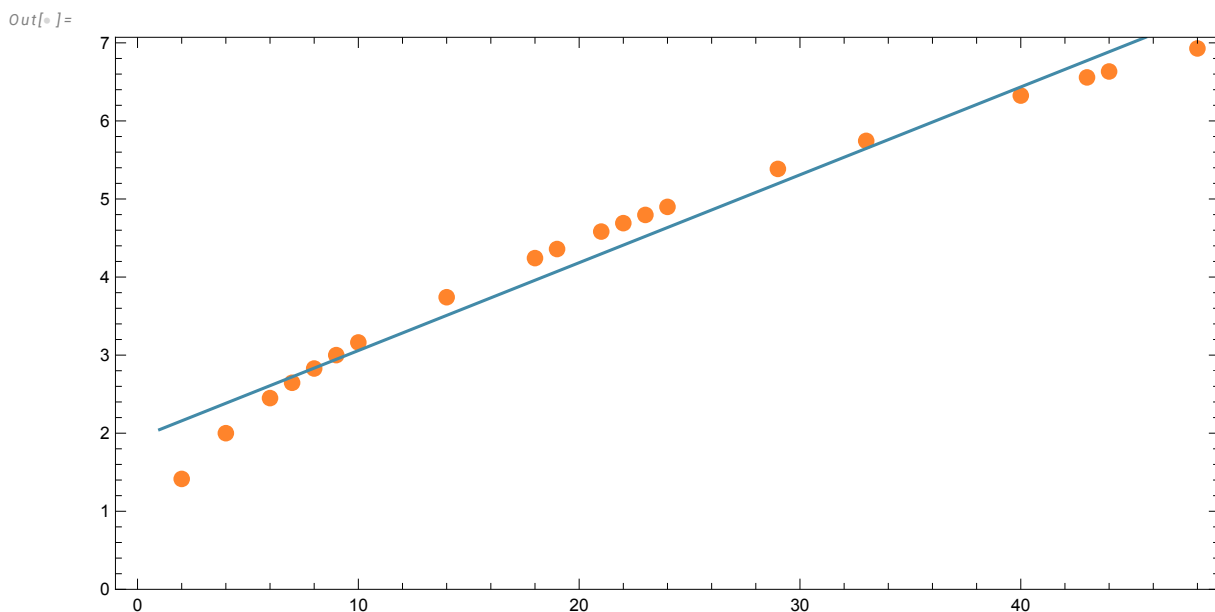


Here we are just assuming a straight line

```

In[ ]:= With[{data = (SeedRandom[34 535];
    {#, Sqrt[#]} & /@
    Sort[RandomSample[Range[50], 20]])},
Show[ListPlot[data, Frame → True,
    PlotRange → {0, Sqrt[50]}, PlotStyle →
    {ChatTechColors["Data"]["Orange"],
    PointSize[.015]}, AspectRatio → 1 / 2],
Plot[Evaluate[Fit[data, {1, x}, x],
    {x, 1, 50},
    PlotRange → {0, Sqrt[50]},
    PlotStyle →
    {ChatTechColors["Data"]["Teal"]}]]]

```



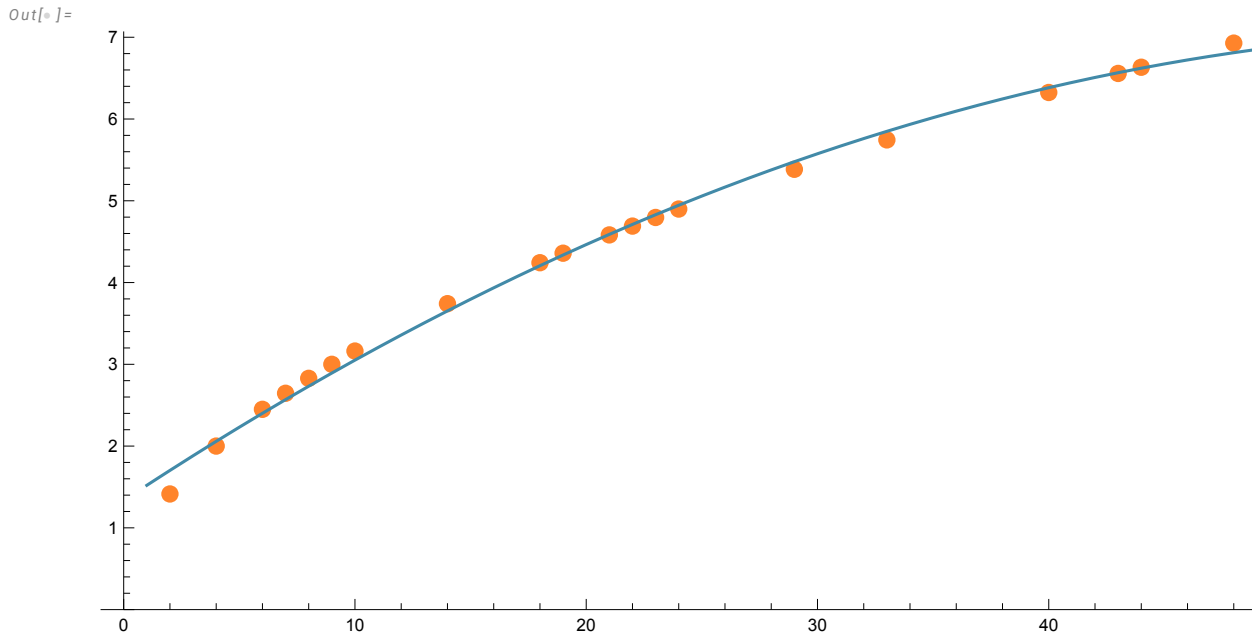
This below actually a better match though. By using a quadratic

curve.

```

In[ ]:= With[{data = (SeedRandom[34 535];
    {#, Sqrt[#]} & /@
    Sort[RandomSample[Range[50], 20]])},
Show[ListPlot[data, Frame → True,
    PlotRange → {0, Sqrt[50]}, PlotStyle →
    {ChatTechColors["Data"] ["Orange"],
    PointSize[.015]}, AspectRatio → 1 / 2],
Plot[Evaluate[Fit[data, {1, x, x^2}, x],
    {x, 1, 50}],
    PlotRange → {0, Sqrt[50]},
    PlotStyle →
    {ChatTechColors["Data"] ["Teal"]}
]]]

```



However, with language there is no mathematical formula that we can use to solve our problem. So this is where we need to approach the solution by looking at how humans have solved problems where there is no one function.

LET'S LOOK AT COMPUTER VISION! It can provide some details. Here's some easy numbers to deal with looking at each pixel and making a prediction.

In[] := **Append**[


Table[**DigitArrayPlot**[n], {n, 0, 5}], "..."]

Out[] =

{ **0**, **1**, **2**, **3**, **4**, **5**, ... }

```
In[ ]:= DigitArrayPlot [Style[4, FontFamily → #]] & /@
{"Bodoni 72", "Didot", "Fira Sans",
 "Gotham", "Papyrus", "ComicSans"}

Out[ ]=
```



Neural Nets were conceived back in the 1940's. Seem to be a simple idealization of how we think the human brain works. Each neuron uses electrical signals to pass information from our photoreceptors and then passed through the network for us to "recognize" an object. Nothing special about LeNet trained on MNIST data. It was constructed back in 1998 and found to work. Used by post office, banks, etc for handwritten recognition of values.

Computers are good at seeing things that it has seen before. They all match here. We are just using a basic neural net and hand written data set from MNIST. MNIST Database of Handwritten Digits, consisting of 60,000 grayscale images of size 28x28.

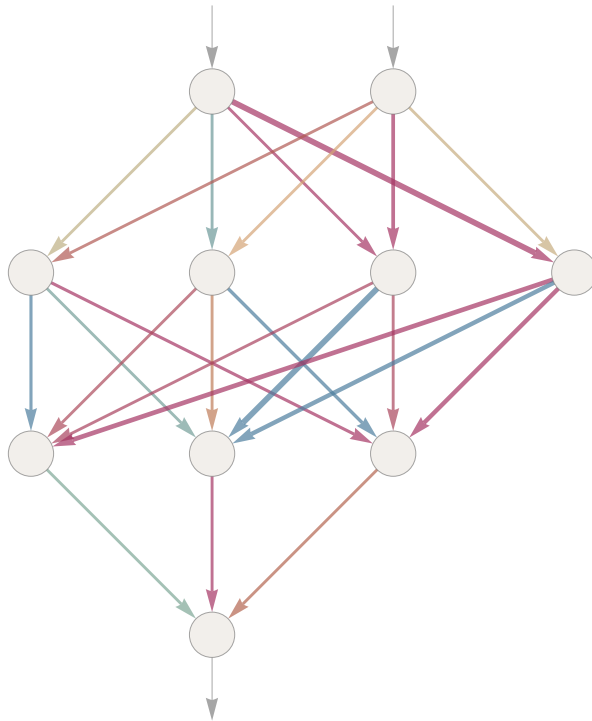
```
In[64]:= NetModel["LeNet Trained on MNIST Data"] [
{7, 0, 9, 7, 8, 2, 4, 1, 1, 1}]
```

```
Out[64]=
{7, 0, 9, 7, 8, 2, 4, 1, 1, 1}
```

Here's a picture of a Neural Network.

```
In[ ]:= NetGraphPlot [ PointRankData2D ["Result431"],
  "AddArrows" → True ]
```

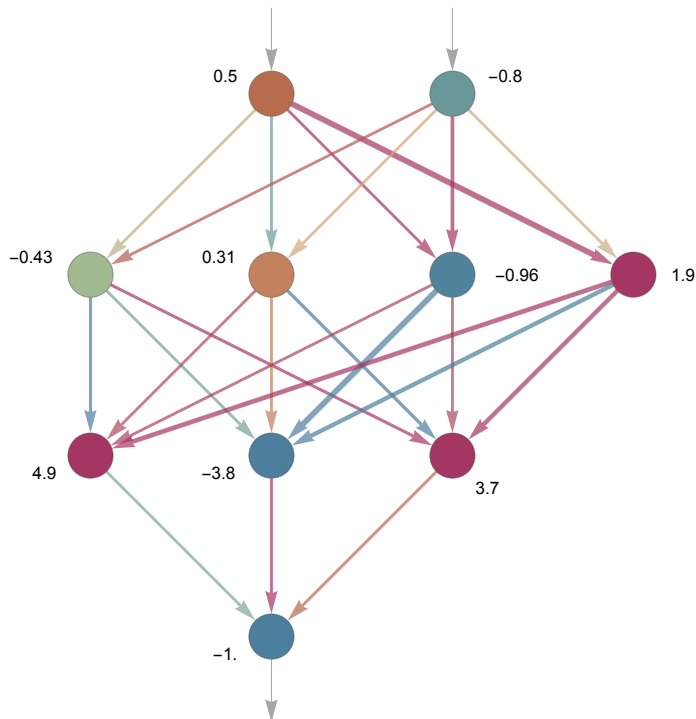
Out[]:=



These are examples of weights being passed through the neural network. Each is tuned and adjusted over time as it learns. The value of a given neuron is determined by multiplying the values of “previous neurons” by their corresponding weights, then adding these up and adding a constant—and finally applying a “thresholding” (or “activation”) function.

```
NetGraphPlot [ PointRankData2D ["Result431"],
  { .5, -.8 }, "AddArrows" → True ]
```

Out[] =

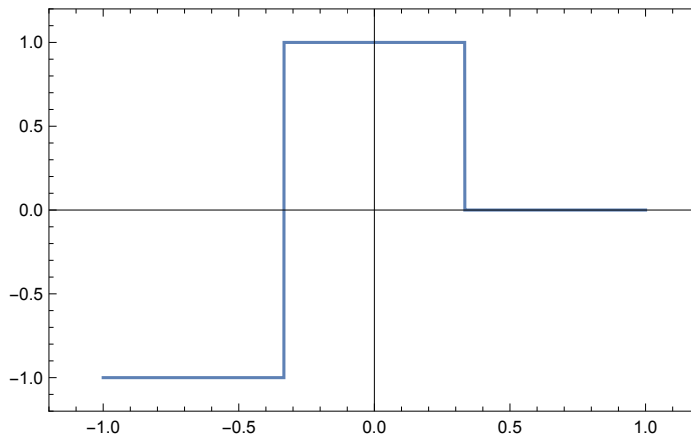


How Does a Neural Net Work?

Here's a very simple function. The computer doesn't know this though!


```
In[*]:= Plot[Piecewise[{{-1, x < -1 / 3},  
    {1, -1 / 3 < x < 1 / 3}, {0, x > 1 / 3}}],  
    {x, -1, 1}, Exclusions → None,  
    Frame → True, PlotRangePadding → .2]
```

Out[*]=



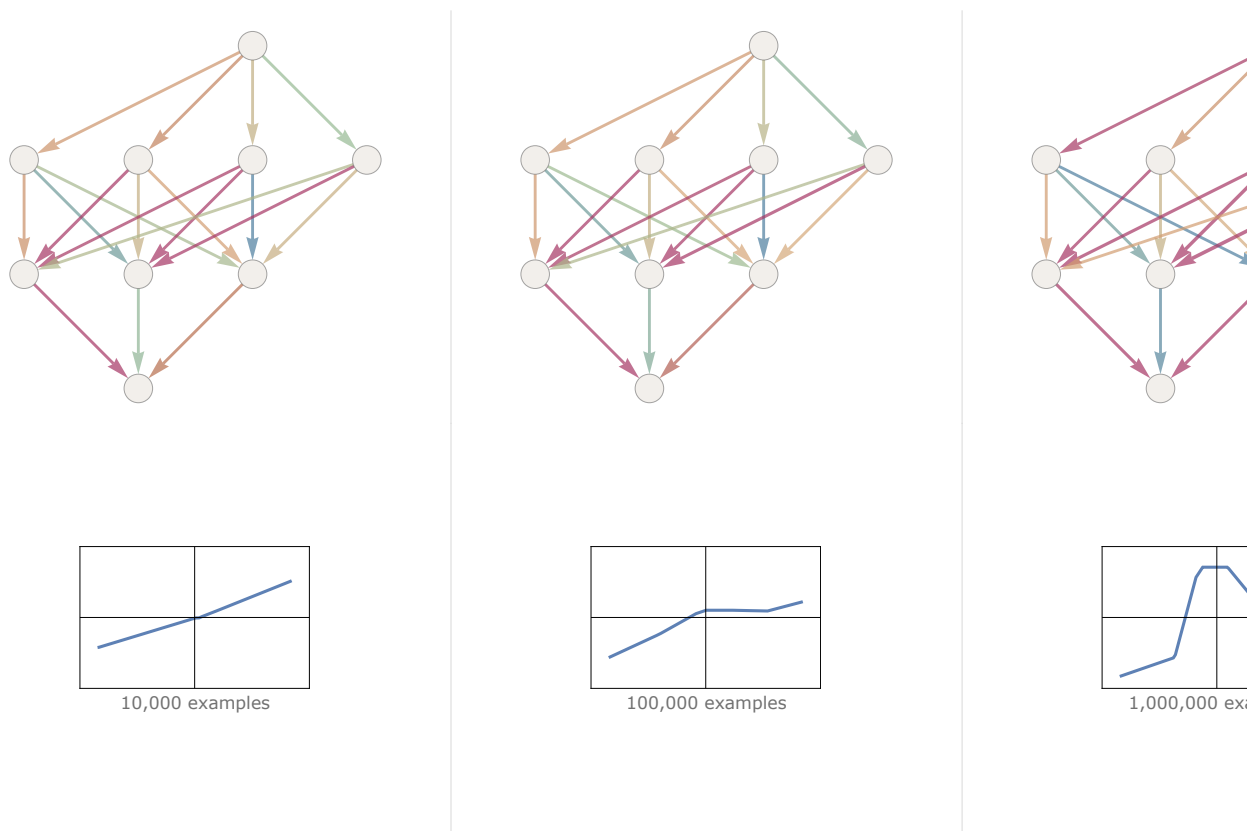
Let's try and have an empty neural network try and figure it out.

```

In[ ]:= GraphicsGrid[
  {
    CompositeLossPlot[
      PiecewiseData1D
    ], ImageSize -> 1000,
    Dividers -> {{False, {True}, False}},
    FrameStyle -> LightGray,
    Spacings -> {45, Automatic}
  ]

```

Out[]:=



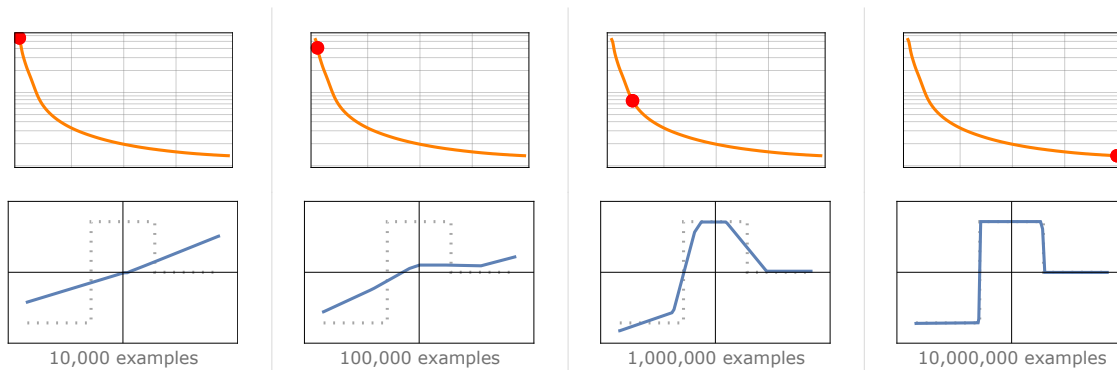
This is how we try to minimize the loss function and finally with the dots it gets closer and closer as we tweak the weights to reduce the loss and get closer to where we want to get closer to the right answer.

```

In[ ]:= GraphicsGrid[
  {
    CompositeLossPlot[
      PiecewiseData1D, "ShowLoss" → True,
      ImageSize → 600, Alignment → Center,
      Dividers → {{False, {True}, False}},
      FrameStyle → LightGray, Spacings → {15, -10}
    ]
  }
]

Out[ ]:=

```



Even though Neural Nets were actually originally thought about in the 40's. It wasn't until the 80's when we started to see layering of neural nets. Then GPU's in 2012 where we could finally do training and use deep neural nets.

IMPORTANT TO NOTE: Ultimately, every neural net just corresponds to some overall mathematical function—though it may be messy to write out. The neural net of ChatGPT also just corresponds to a mathematical function like this—but effectively with BILLIONS of terms. For the example above, it would be:

```
In[ ]:= NetToFunction [ PointRankData2D [
  "Result431"], {x, y}, {w, b} ] [-1, 1] /.

```

Ramp → f // TraditionalForm

Out[]//TraditionalForm=

$$\begin{aligned}
 &w_{511}f(w_{311}f(b_{11} + x w_{111} + y w_{112}) + \\
 &\quad w_{312}f(b_{12} + x w_{121} + y w_{122}) + \\
 &\quad w_{313}f(b_{13} + x w_{131} + y w_{132}) + \\
 &\quad w_{314}f(b_{14} + x w_{141} + y w_{142}) + b_{31}) + \\
 &w_{512}f(w_{321}f(b_{11} + x w_{111} + y w_{112}) + \\
 &\quad w_{322}f(b_{12} + x w_{121} + y w_{122}) + \\
 &\quad w_{323}f(b_{13} + x w_{131} + y w_{132}) + \\
 &\quad w_{324}f(b_{14} + x w_{141} + y w_{142}) + b_{32}) + \\
 &w_{513}f(w_{331}f(b_{11} + x w_{111} + y w_{112}) + \\
 &\quad w_{332}f(b_{12} + x w_{121} + y w_{122}) + \\
 &\quad w_{333}f(b_{13} + x w_{131} + y w_{132}) + \\
 &\quad w_{334}f(b_{14} + x w_{141} + y w_{142}) + b_{33}) + b_{51}
 \end{aligned}$$

How Do We Train a Neural Net?





















































































































Let's try and train our own on the hand written text.

What happens with ChatGPT is that you can look at a series of words and then mask out the following words and see how close ChatGPT was to getting it right. You then train the model and tweak the weights to get it closer to the right answer.

Look at dataset from MNIST - written text of about 10,000 integers. Here we take 500 samples and train on it.

In[56]:= RandomSample[ResourceData["MNIST"], 500]

Out[56]=

{  → 6,  → 9,  → 4,  → 2,  → 4,
 → 5,  → 4,  → 7,  → 4,  → 8,
 → 6,  → 8,  → 5,  → 7,  → 8,
 → 5,  → 3,  → 4,  → 5,  → 7,
 → 4,  → 0,  → 0,  → 7,  → 8,  → 5,
 → 1,  → 3,  → 7,  → 5,  → 9,  → 7,
 → 5,  → 5,  → 2,  → 0,  → 6,  → 0,
 → 5,  → 3,  → 8,  → 4,  → 7,  → 5,
 → 4,  → 3,  → 1,  → 4,  → 3,  → 8,
 → 6,  → 1,  → 3,  → 3,  → 6,  → 8,
 → 8,  → 5,  → 7,  → 7,  → 3,  → 2,
 → 6,  → 0,  → 9,  → 3,  → 3,  → 1,
 → 1,  → 7,  → 3,  → 4,  → 9,  → 1,
 → 4,  → 9,  → 1,  → 1,  → 3,  → 0,
 → 7,  → 5,  → 9,  → 7,  → 2,  → 9,
 → 8,  → 3,  → 1,  → 1,  → 7,  → 0,
 → 1,  → 9,  → 4,  → 9,  → 2,  → 9,
 → 2,  → 8,  → 5,  → 7,  → 7,  → 5,
 → 7,  → 3,  → 1,  → 8,  → 9,  → 9,
 → 1,  → 7,  → 9,  → 1,  → 1,  → 0,

9 → 9, 3 → 3, 6 → 6, 5 → 5, 5 → 5, 5 → 5,
 6 → 6, 1 → 1, 3 → 3, 5 → 5, 8 → 8, 4 → 4,
 8 → 8, 1 → 1, 0 → 0, 1 → 1, 1 → 1, 7 → 7,
 4 → 4, 9 → 9, 1 → 1, 3 → 3, 1 → 1, 6 → 6,
 5 → 5, 2 → 2, 8 → 8, 0 → 0, 8 → 8, 8 → 8,
 8 → 8, 1 → 1, 8 → 8, 4 → 4, 9 → 9, 1 → 1,
 2 → 2, 3 → 3, 4 → 7, 8 → 8, 8 → 8, 9 → 9,
 7 → 7, 8 → 8, 0 → 0, 0 → 0, 1 → 1, 9 → 9,
 2 → 2, 5 → 5, 0 → 0, 8 → 8, 4 → 4, 5 → 5,
 5 → 5, 3 → 3, 0 → 0, 5 → 5, 8 → 8, 1 → 1,
 1 → 1, 5 → 5, 4 → 4, 8 → 8, 7 → 7, 9 → 9,
 9 → 9, 4 → 4, 2 → 2, 1 → 1, 2 → 2, 9 → 9,
 0 → 0, 6 → 6, 7 → 7, 1 → 1, 1 → 1, 8 → 8,
 4 → 4, 5 → 5, 8 → 8, 9 → 9, 4 → 4, 1 → 1,
 2 → 2, 7 → 7, 0 → 0, 1 → 1, 0 → 0, 3 → 3,
 3 → 3, 1 → 1, 7 → 7, 1 → 1, 1 → 1, 2 → 2,
 0 → 0, 8 → 8, 4 → 4, 1 → 1, 8 → 8, 2 → 2,
 0 → 0, 8 → 8, 2 → 2, 1 → 1, 9 → 9, 9 → 9,
 5 → 5, 9 → 9, 6 → 6, 3 → 3, 9 → 9, 2 → 2,
 3 → 3, 4 → 4, 9 → 9, 2 → 2, 0 → 0, 9 → 9,
 1 → 1, 5 → 5, 7 → 7, 1 → 1, 0 → 0, 0 → 0,

4 → 4, 4 → 4, 9 → 9, 1 → 1, 9 → 9, 6 → 6,
 0 → 0, 0 → 0, 0 → 0, 1 → 1, 0 → 0, 2 → 2,
 9 → 9, 1 → 1, 8 → 8, 1 → 1, 6 → 6, 8 → 8,
 4 → 4, 9 → 9, 3 → 3, 0 → 0, 1 → 1, 0 → 0,
 8 → 8, 9 → 9, 6 → 6, 3 → 3, 1 → 1, 0 → 0,
 6 → 6, 2 → 2, 0 → 0, 8 → 8, 9 → 9, 1 → 1,
 7 → 7, 9 → 9, 2 → 2, 5 → 5, 9 → 9, 7 → 7,
 7 → 7, 4 → 4, 6 → 6, 2 → 2, 2 → 2, 2 → 2,
 5 → 5, 0 → 0, 6 → 6, 8 → 8, 2 → 2, 0 → 0,
 5 → 5, 9 → 9, 9 → 9, 4 → 4, 9 → 9, 5 → 5,
 2 → 2, 7 → 7, 6 → 6, 6 → 6, 3 → 3, 8 → 8,
 4 → 4, 0 → 0, 2 → 2, 7 → 7, 9 → 9, 3 → 3,
 7 → 7, 4 → 4, 8 → 8, 2 → 2, 4 → 4, 6 → 6,
 5 → 5, 5 → 5, 4 → 4, 9 → 9, 2 → 2, 4 → 4,
 0 → 0, 5 → 5, 4 → 4, 2 → 2, 9 → 9, 3 → 3,
 4 → 4, 7 → 7, 7 → 7, 2 → 2, 0 → 0, 8 → 8,
 7 → 7, 8 → 8, 0 → 0, 0 → 0, 3 → 3, 9 → 9,
 5 → 5, 6 → 6, 1 → 1, 2 → 2, 1 → 1, 7 → 7,
 1 → 1, 4 → 4, 1 → 1, 9 → 9, 0 → 0, 8 → 8,
 9 → 9, 1 → 1, 6 → 6, 9 → 9, 4 → 4, 8 → 8,
 7 → 7, 7 → 7, 0 → 0, 7 → 7, 4 → 4, 7 → 7,

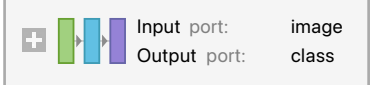
1 → 2, 9 → 9, 0 → 0, 9 → 9, 1 → 1, 0 → 0,
 0 → 0, 3 → 3, 8 → 8, 6 → 6, 0 → 0, 2 → 2,
 9 → 9, 4 → 4, 8 → 8, 1 → 1, 5 → 5, 9 → 9,
 7 → 7, 6 → 6, 0 → 0, 5 → 5, 2 → 2, 1 → 1,
 2 → 2, 7 → 7, 6 → 6, 6 → 6, 2 → 2, 4 → 4,
 9 → 9, 4 → 4, 5 → 5, 2 → 2, 9 → 9, 1 → 1,
 7 → 7, 9 → 9, 3 → 3, 9 → 9, 5 → 5, 8 → 8,
 0 → 0, 8 → 8, 8 → 8, 0 → 0, 1 → 1, 9 → 9,
 7 → 7, 6 → 6, 3 → 3, 2 → 2, 2 → 2, 5 → 5,
 1 → 1, 0 → 0, 1 → 1, 2 → 2, 1 → 1, 8 → 8,
 0 → 0, 1 → 1, 9 → 9, 3 → 3, 2 → 2, 2 → 2,
 7 → 7, 7 → 7, 4 → 4, 3 → 3, 9 → 9, 9 → 9,
 6 → 6, 0 → 0, 7 → 7, 2 → 2, 9 → 9, 6 → 6,
 6 → 6, 4 → 4, 7 → 7, 1 → 1, 8 → 8, 4 → 4,
 7 → 7, 3 → 3, 5 → 5, 4 → 4, 5 → 5, 1 → 1,
 1 → 1, 1 → 1, 9 → 9, 1 → 1, 5 → 9, 7 → 7,
 6 → 6, 2 → 2, 4 → 4, 2 → 2, 1 → 1, 8 → 8,
 7 → 7, 8 → 8, 9 → 9, 4 → 4, 2 → 2, 9 → 9,
 1 → 1, 0 → 0, 3 → 3, 8 → 8, 5 → 5, 3 → 3,
 2 → 2, 0 → 0, 8 → 8, 7 → 7, 6 → 6, 8 → 8,
 3 → 3, 7 → 7, 7 → 7, 9 → 9, 4 → 4, 3 → 3,

$1 \rightarrow 1, 5 \rightarrow 5, 6 \rightarrow 6, 7 \rightarrow 7, 3 \rightarrow 3, 6 \rightarrow 6\}$

Train our own NeuralNet and watch the loss going down. Super simple example of how to train an NeuralNet with 2000 examples and then predict the text. Gets it at 100%

```
In[57]:= NetTrain[NetModel["LeNet"], %]
```

```
Out[57]=
```

NetChain []

```
In[58]:= % [ 4 ]
```

```
Out[58]=
```

4

Let's look at the probabilities for the answer of each letter. 0 element is 0, next is 1, etc.

```
In[59]:= SoftmaxLayer[] [Drop[%20, -1] [ 4 ]]
```

```
Out[59]=
```

$\{7.65215 \times 10^{-9}, 1.28519 \times 10^{-10},$
 $3.66833 \times 10^{-8}, 6.8442 \times 10^{-7}, 0.999962,$
 $2.75135 \times 10^{-9}, 8.29109 \times 10^{-8},$
 $9.34436 \times 10^{-7}, 5.07073 \times 10^{-7}, 0.000035274\}$

Let's back up 1 layer in the neural net and see where things are at before the final decision. Look at this layer. They show the “4-ness” of what kind of thing we are seeing. And we can keep shifting back more and more layers.

```
In[*]:= Drop[%55, -1] [ 4 ]
```

```
Out[*]=
```

```
{-5.80964, -14.6164, -4.01125,
 -11.6114, 15.2185, -1.36717,
 0.379412, 2.30428, -2.44465, 4.98982}
```

This is the key to **attention**! When we “nailed the task” we can backup and look and see what was the signature we can see earlier in the network the signature. See below how these images live in dimensional space.

There is something going on with putting these into dimensional space with each pixel in the image.

```

In[60]:= FeatureSpacePlot3D[SeedRandom[123];
  # → SetAlphaChannel[#, ColorNegate[#]] & @@@
  RandomSample[ResourceData["MNIST"], 50],
  FeatureExtractor → NetDrop[NetModel[
    "LeNet Trained on MNIST Data"], -3]]

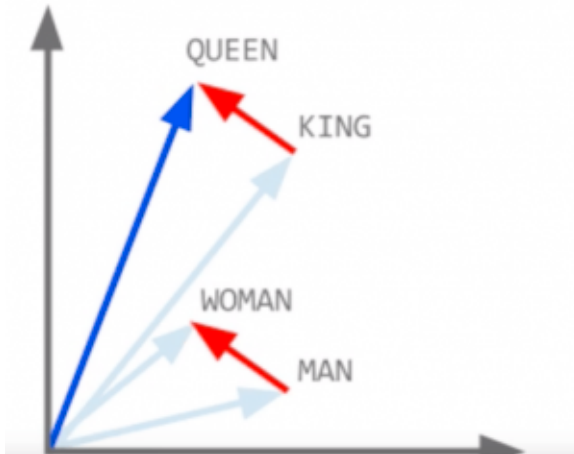
```

Out[60]=

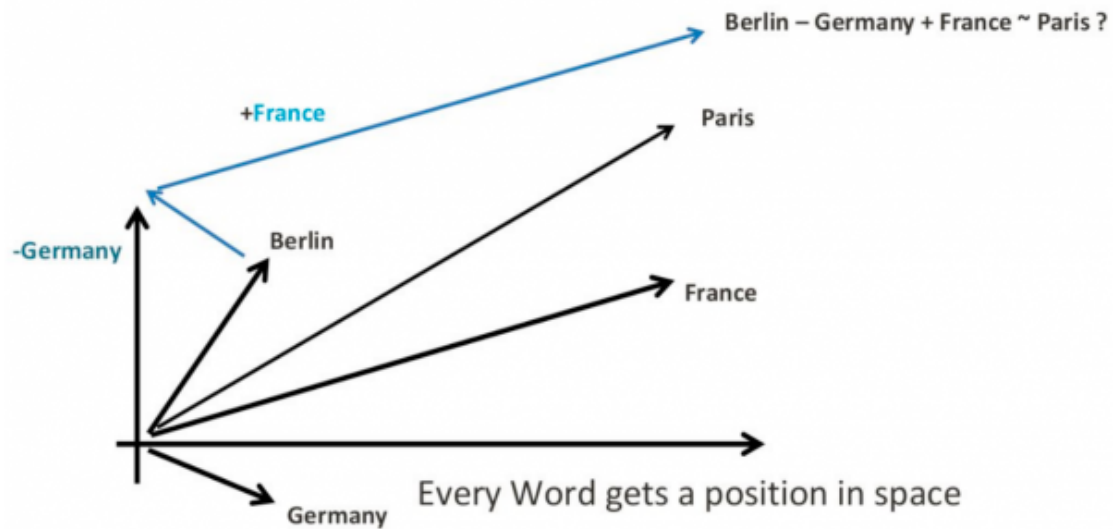


Can we do this with words? Turns out that YES we can! We do this with “Embeddings”. Words that have a similar meaning are put in to the same dimensional space. This is how we represent words with numbers. This is just 2 dimensional. Roughly though the same thing we did with the images of numbers above.

Let’s recall all we are trying to do is predict the next word.



Can also derive meaning for example (Berlin - Germany + France = Paris)

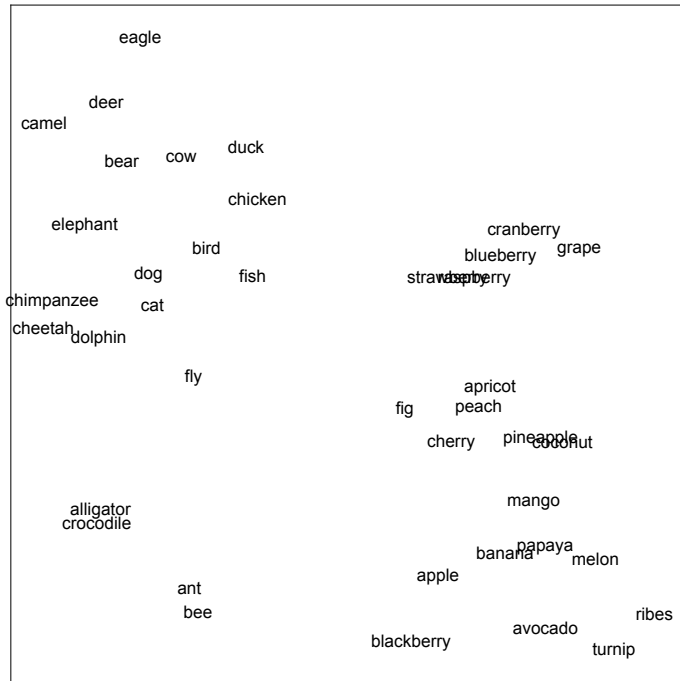


```

In[ ]:= FeatureSpacePlot[
  {"alligator", "ant", "bear", "bee", "bird",
   "camel", "cat", "cheetah", "chicken",
   "chimpanzee", "cow", "crocodile",
   "deer", "dog", "dolphin", "duck",
   "eagle", "elephant", "fish", "fly",
   "apple", "apricot", "avocado", "banana",
   "blackberry", "blueberry", "cherry",
   "coconut", "cranberry", "grape", "turnip",
   "mango", "melon", "papaya", "peach",
   "pineapple", "raspberry", "strawberry",
   "ribes", "fig"}, FeatureExtractor →
  NetModel["GloVe 300-Dimensional
            Word Vectors Trained on
            Wikipedia and Gigaword 5 Data"],
  Frame → True, FrameTicks → None]

```

```
Out[ ] =
```



Computers work much better with numbers. Embeddings are just a number representing word pairs. Here's an example below. You can assign each word in the 50,000 words a unique number. However, we need to store them in dimensional space. Each word is represented as a matrix.

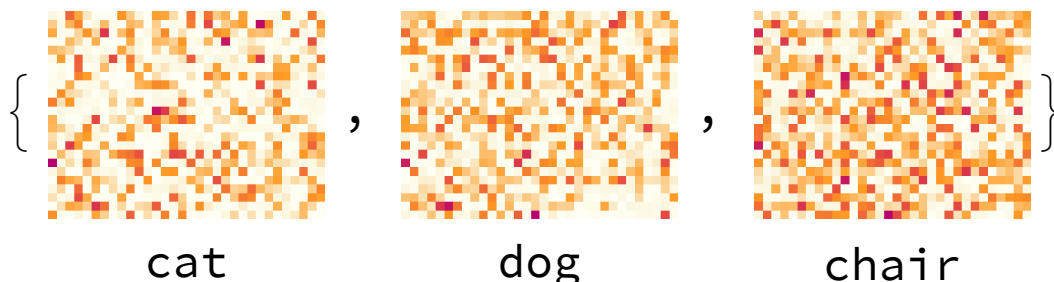
We can look at the feature vectors and projecting them to 2-dimensional space to see how similar they are. We are looking at what makes a “cat” and “dog” be a choice by the network vs using the word “chair”. **Can't tell that cat and dog are close to each other, but a computer can.** NOTE: ChatGPT uses chunks of words, rather than specific words, but the concept is the same.

```

In[ ]:= Labeled[ArrayPlot[Partition[
    First[NetExtract[
        NetModel["GPT2 Transformer Trained
        on WebText Data"],
        {"embedding", "embeddingtokens"}][
    NetExtract[NetModel[
        "GPT2 Transformer Trained on
        WebText Data"],
        "Input"][#]]], 32], Frame →
    None, ImageSize → 150,
    ColorFunction →
        ChatTechColors [
            "Orange"]],
    #] & /@ {"cat", "dog",
    "chair"}

```

Out[]=



Embedding Vector Example:

CAT =

```
{{-0.0802898,0.0340174,-0.49889,0.0117027,0.019475,-0.174645,0
```

.384173,-0.0108258,-0.184527,0.127755,..}}

DOG =

{{-0.102029,-0.0657201,-0.32943,-0.0616078,-0.00910606,-0.160064,0.317742,-0.202682,-0.178742,0.011637,..}}

CHAIR =

{{-0.145438,-0.123598,-0.279552,-0.0516923,-0.0281756,-0.307468,1.21536,-0.190257,-0.216411,...}}

But actually we can go further than just characterizing words by collections of numbers; we can also do this for sequences of words, or indeed whole blocks of text. Inside ChatGPT that's how it's dealing with things.

1. It takes the text it's got so far, and generates an embedding vector to represent it.
2. Then its goal is to find the probabilities for different words that might occur next by feeding it through the neural network.
3. After generating the next token, that value is added to the prior token.
4. Go back to #1

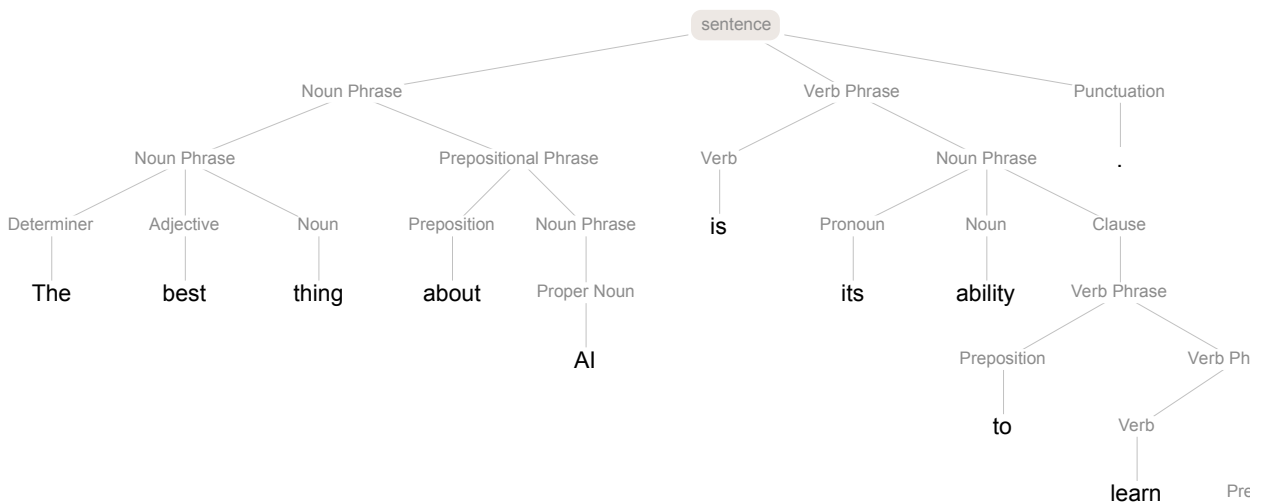
Using these embeddings and previous passes it knows which words should be paid attention. This is interesting since **the neural net starts to learn the underlying structure of what language is!**

We do know that human language has structure. ChatGPT has actually been able to show that there is more regularity than we

thought was there and has learned it! ChatGPT has learned syntactic grammar.

```
In[*]:= First[TextStructure[
  "The best thing about AI is its
  ability to learn from experience.",
  "ConstituentGraphs"]]
```

Out[*]=



ChatGPT has learned meaningful sequences. Patterns of word usage and you could substitute in different words to come to the same conclusion. People say that ChatGPT is “figuring it out”, but **it’s really just using similar word placement because it’s seen that it can.** It’s following a templated structure of logic.

Let’s See How ChatGPT is Moving Through The Sentence

This graph shows how it moves around and hops from one to another.

```

In[ ]:= Module[{step, promptcolor,
  color, result, vocabulary, i = 1},
vocabulary =
  NetExtract[NetExtract[NetModel[
    {"GPT2 Transformer Trained on WebText
      Data",
    "Task" → "LanguageModeling"}],
    "Output"], "Labels"];

result = ;

promptcolor =
  ;
color = ;
step = result["PromptLength"] +
  Replace[i, {Automatic →
    result["ContinuationLength"],
  n_Integer ⇒ Clip[n, {1,
    result["ContinuationLength"]}]}];

Show[
  ListPlot[{Thread[Callout[result["Points"]]]

```

```

1 ;; result["PromptLength"], -1]],
result["Tokens"][[
1 ;; result["PromptLength"]]]]],
Thread[Callout[result["Points"][[result[
"PromptLength"] + 1 ;; step, -1]],
result["Tokens"][[result[
"PromptLength"] + 1 ;; step]]]],
Thread[Callout[
result["Points"][[step, 1 ;; -2]],
Thread[Style[
vocabulary[[result["BestPositions"][[
step, 1 ;; -2]]],
Opacity[0.5]]]]]],
PlotStyle → {promptcolor, color,
Directive[Black, Opacity[0.3]]}},
Graphics[{{PointSize[0.01], {promptcolor,
Point[result["PromptPoints"][[1]]}},
{color, Point[
result["Points", step - 1, -1]]}},
Point[result["Points", step, -1]]}},
{{promptcolor, AbsoluteThickness[1.5],
Line[result["PromptPoints"]]},
{color, AbsoluteThickness[2],

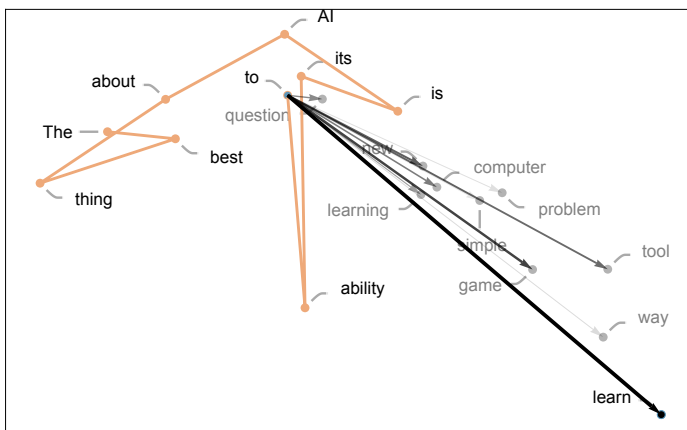
```

```

Line[Take[result[["Points"], All, -1],
      {result["PromptLength"],
       step - 1}]]],
Arrowheads[0.02],
MapThread[{Opacity[#2 + 0.1],
           AbsoluteThickness[2 * #2],
           Arrow[{result[["Points"], step - 1, -1],
                 #1}]] &, {result[["Points"], step],
Rescale[
           result[["Probabilities", step]]}],
PlotRange → Automatic, ImageSize →
Medium,
Frame → True, FrameTicks → False,
Axes → None, PlotRange → result["Range"],
PlotRangePadding → Scaled[0.05]]]

```

Out[]:=



Here is how it “fans out” as it goes from each word to word

In[]:= **With**[

```

{fun = Function[Module[{step, promptcolor,
    color, result, i = #},
    promptcolor =
      ChatTechColors["Normal"] [.2];
    color = RGBColor[{15, 162, 127} / 255];
    result = The best thing about AI is its ability to...;
    step = result["PromptLength"] +
      Replace[i, {Automatic →
        result["ContinuationLength"]
      },
      n_Integer ⇒ Clip[n, {1, result[
        "ContinuationLength"]}]]];
    Labeled[Show[

Graphics[{{PointSize[0.01],
    {promptcolor, Point[result[
      "PromptPoints"][[1]]}},
    {color, Point[
      result["Points", step - 1, -1]]},
    Point[
      result["Points", step, -1]]},
    {{promptcolor, AbsoluteThickness[

```

```

1.5], Line[result[
  "PromptPoints"]]}, {color,
AbsoluteThickness[2], Line[
  Take[result["Points", All, -1],
    {result["PromptLength"],
      step - 1}]
]}}], Arrowheads[0.02],
MapThread[{Opacity[#2 + 0.1],
  AbsoluteThickness[
    2 * #2], Arrow[{result[
      "Points", step - 1, -1],
    #1}]} &, {result["Points",
step], Rescale[result[
  "Probabilities", step]]}],
PlotRange → result["Range"],
PlotRangePadding →
  Scaled[0.05], Frame
  → True, FrameTicks → None,
AspectRatio → 1 / GoldenRatio
],
ListPlot[
{Callout[result["Points"]][step,

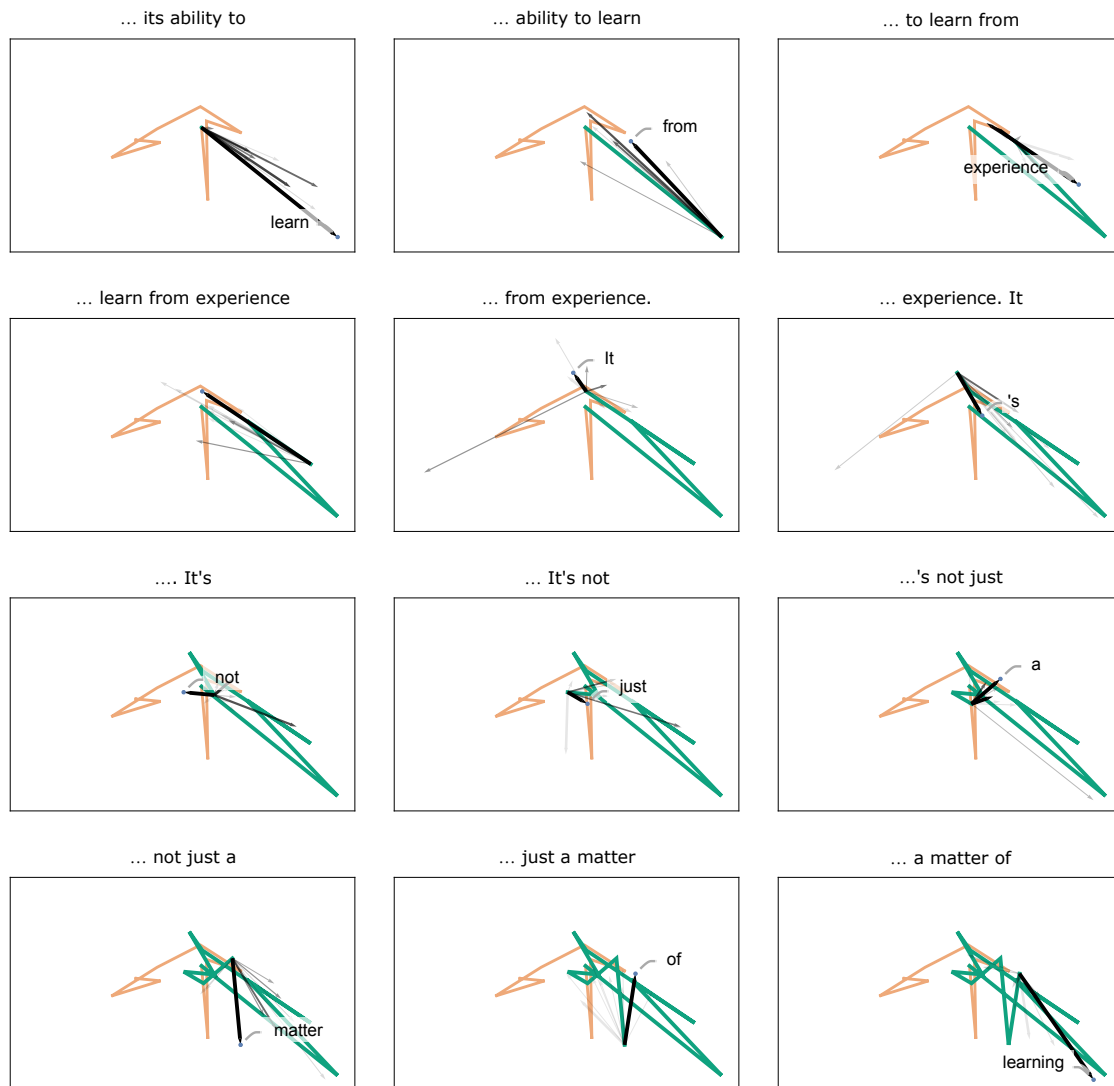
```

```

-1]], result["Tokens"][[step]],
PlotStyle →
ChatTechColors["Normal"] [.5]},
PlotRange → result["Range"]
], Style["..." <> result["Tokens"][[
step - 3 ;; step - 1]], "Label"], Top]
]],},
GraphicsGrid[
Partition[Map[fun, Range[12]], 3],
ImageSize → 600, Spacings → {0, 20}]
]

```

Out[]:=



Finally we can then look at it in 3D space. Green is what it chose, but grey is what it considered. As you move through words you are moving through space.

```
In[ ]:= With[{resultsGPT2 = The best thing about AI is its ability to... + },
Graphics3D[{{AbsoluteThickness[2],
```



```

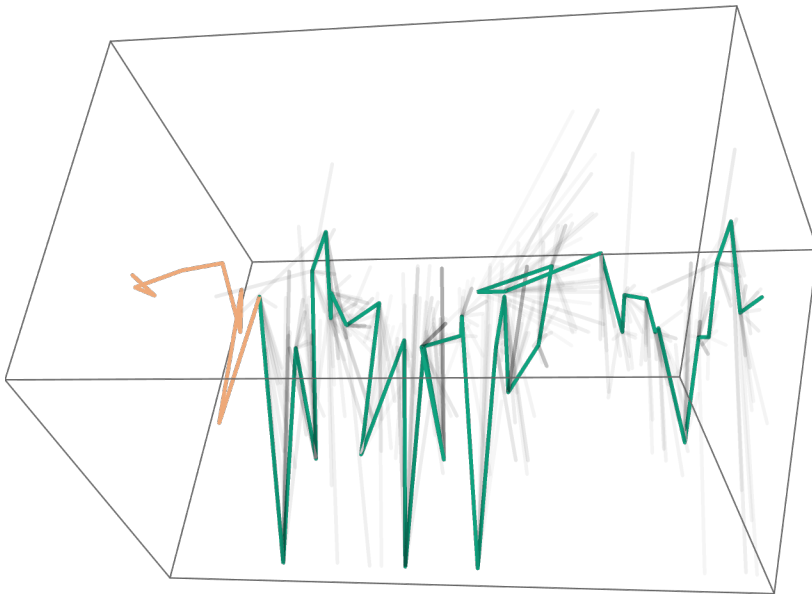
[■] ChatTechColors ["Normal"] [0.2],
Line[MapIndexed[Append[#1, First[#2]] &,
  resultsGPT2[["PromptPoints"]]]],
Table[{Black,
  Table[{AbsoluteThickness[2],
    Opacity[resultsGPT2[["Probabilities",
      i, j]] + 0.03], Line[
      {Append[resultsGPT2[["Points", i - 1,
        -1]], i - 1], Append[resultsGPT2[["Points", i, j]], i]}], {j,
      resultsGPT2["Continuations"] - 1}]],
  {RGBColor[ $\frac{1}{255}$  {15, 162, 127}],
    AbsoluteThickness[2],
    Line[{Append[resultsGPT2[["Points", i - 1, -1]], i - 1],
      Append[resultsGPT2[["Points",
        i, -1]], i]}]},
  {i, resultsGPT2["PromptLength"] + 1,
    resultsGPT2["PromptLength"] +
      resultsGPT2["ContinuationLength"]}],
BoxRatios → {1, 1, GoldenRatio},

```

ViewPoint →

$\{-1.5,$
 $-3, 2\},$

ViewVertical → $\{-1, 0, 0\} \Big]$



SUMMARY

I've skipped a lot of the underlying complex math - on purpose.

Overview:

1. Reviewed that you can build basic language from just probabilities or letters, spaces and what should follow.
2. Follows then that you can do the same with words (and tokens).
2. Using n-grams only get you so far though.
 - 2a. There is no way to “brute force” the generation of text.
 - 2b. Not enough text to have seen every possible combina-

tion.

3. To get it to work we need to take concepts from Computer Vision

3a. Build a Neural Net

3b. Create word embeddings to model language

3c. Transformer Architecture - allows using self-attention to capture the relationships looking back in the sequence, then feeding it forward through the Neural Net.

In the end:

1. Chat GPT discovered that using computation we can put together sentences and semantic grammar that **it can learn language design**. In the same way we have taught computers to learn Chess and Go.

2. However, in the same way we have taught computers to learn games like Chess and Go, it does better when it's been coaxed. OpenAI added an additional layer that we didn't talk about - Applied Reinforcement Learning with Human Feedback (RLHF) to help steer the answers. These are the "hints". More on that in future OSN talk!