Introduction to Machine Learning in Python

Presented by Nicholas Miklaucic

2024-07-22

Outline

Outline

Machine Learning (ML)

Machine learning (ML) is when a computer does something through some kind of experience, not just a pre-programmed algorithm.. More formally:

> A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.
>
> — Tom Mitchell, **Machine Learning**

Supervised Learning

- Prediction of some *target*, often y, from labeled training data, often X.

There are two broad kinds of supervised learning:

- *Classification* predicts one of a set of discrete *classes*, like diagnosing a disease.
- *Regression* predicts a numerical output, like predicting the price a house will sell at.

Figure 1: Zillow thinks this house is worth $4 million.

Figure 2: A ML model could classify this image as a picture of a dog.

Supervised Learning

Consider this an unofficial exercise 0: what are other examples of classification and regression? What examples might appear in materials science?

Outline

Evaluating ML Models

If we have a model making predictions, how do we assess how good our prediction is?

Let’s call what our model predicts $\hat{y}$ (“y-hat”), and we have the actual y to compare it with.

Classification

We can plot $\hat{y}$ vs. y in a *confusion matrix*. This is basically all of the important information we need to evaluate the model. Sometimes we use the raw counts, sometimes we normalize by row like here.

The most common metric is *accuracy*: the percent of correct predictions. (This is the fraction of values on the main diagonal.)

There are many other metrics that can be better when classes are imbalanced. For now, we'll stick with accuracy, but looking at the full confusion matrix gives a lot more information.

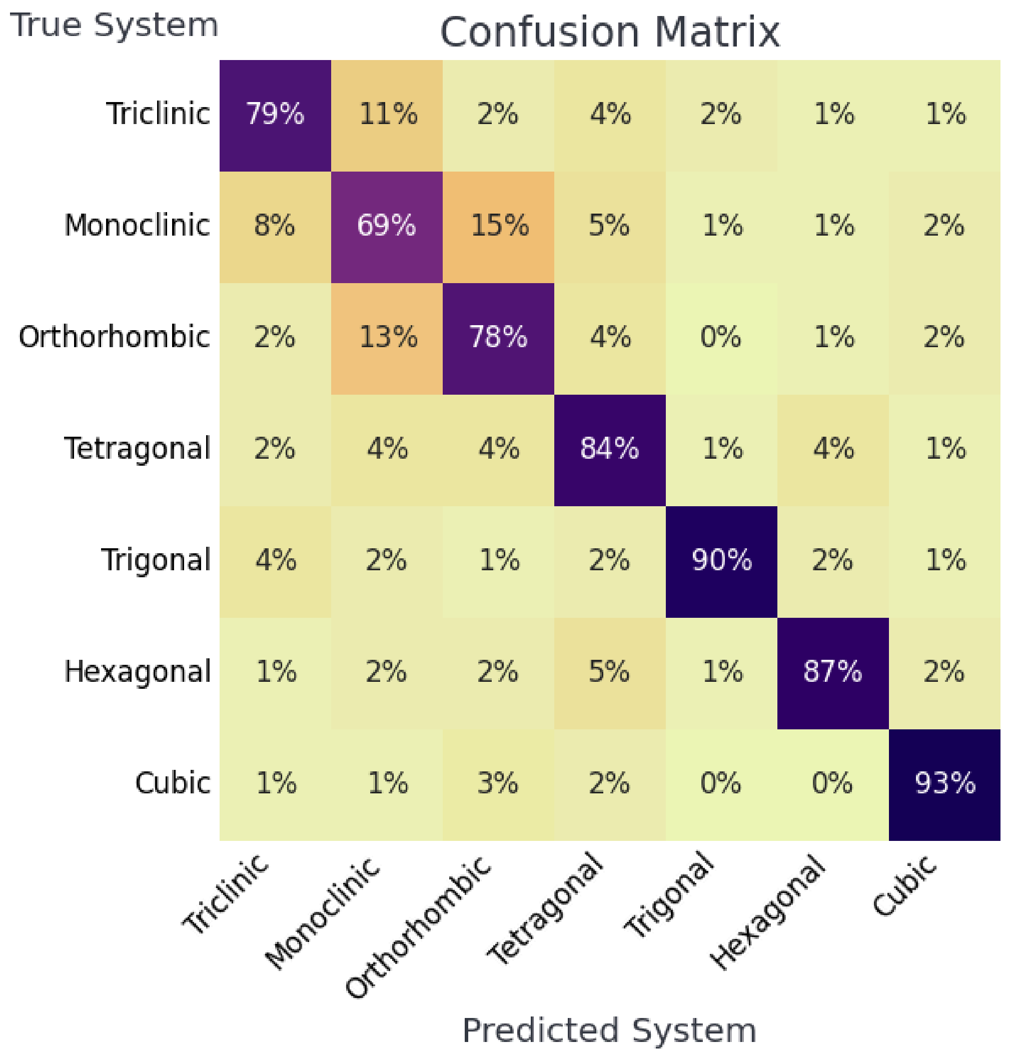


Figure 3: Reproduced from Li, Dong, Yang, & Hu 2021

Regression

We can also plot the predicted vs. actual values, as shown here.

There are a couple common metrics you will see:

Mean Squared Error (MSE): $\text{mean}\left((y - \hat{y})^2\right)$

Root Mean Squared Error (RMSE aka L2): $\sqrt{\text{mean}\left((y - \hat{y})^2\right)}$

Mean Absolute Error (MAE aka L1): $\text{mean}(|y - \hat{y}|)$

R^2: $1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$

MSE is in units of the target squared. RMSE and MAE are in the same units as the target: they're different versions of "average" error. R^2 is unitless: it measures the percent of

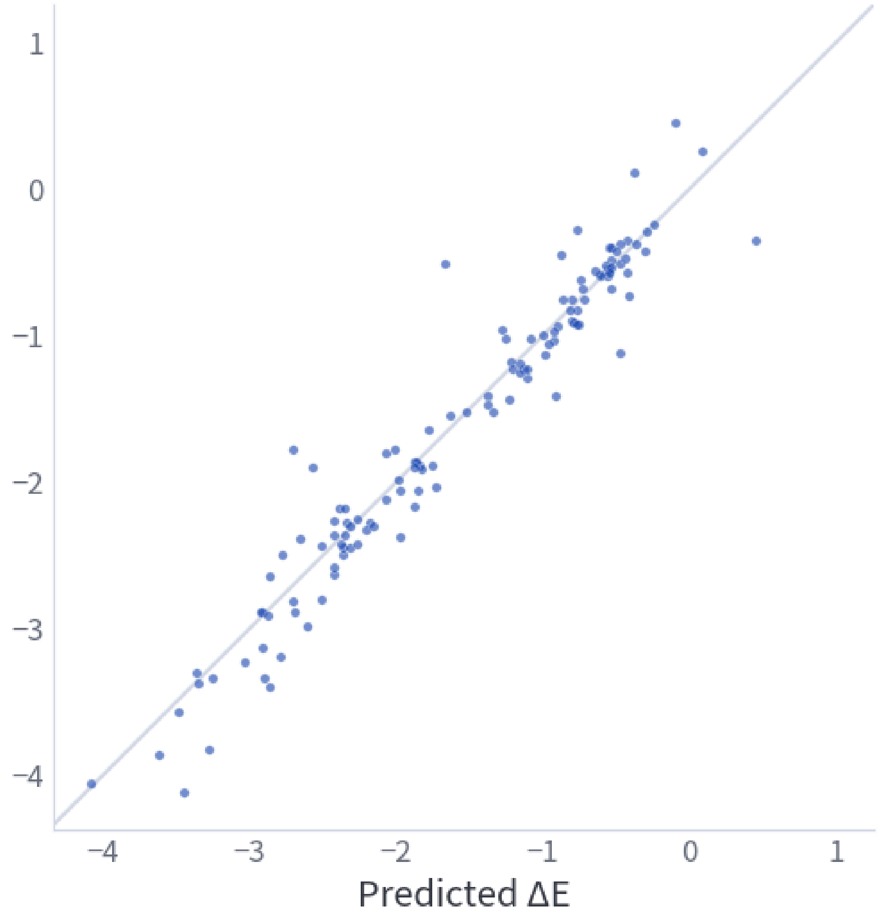


Figure 4: Formation energy prediction

the variance in the target we can explain with the inputs,
so 0 is the same as no information and 1 is perfect.

Validation

We want to know how well our model will do on *new* inputs. If we can compute accuracy or MAE, then we already know those answers! Scientists don't want AI to tell them what they already know.

Moreover, performance on training data isn't predictive of future performance. Models can *overfit*, finding patterns that don't hold up.

We want to see how our model will do on data it hasn't seen before. How can we do that?

Hold-out Set

We can split our data into sets. Let's say we have 100 *samples*.

If we train our model on 80 of those samples, and then test it on the remaining 20, we have a better estimate of our model's future performance.

If we have a ton of data, this works well. If we don't have enough data, we face a tricky dilemma:

- If we use too much data for validation, our model won't have enough training data, and our performance will be much lower than it would be had we trained on the whole set.
- If we use too little data for validation, the estimates will be unreliable, and randomness in how our validation set is chosen will end up dominating the results.

Cross-Validation

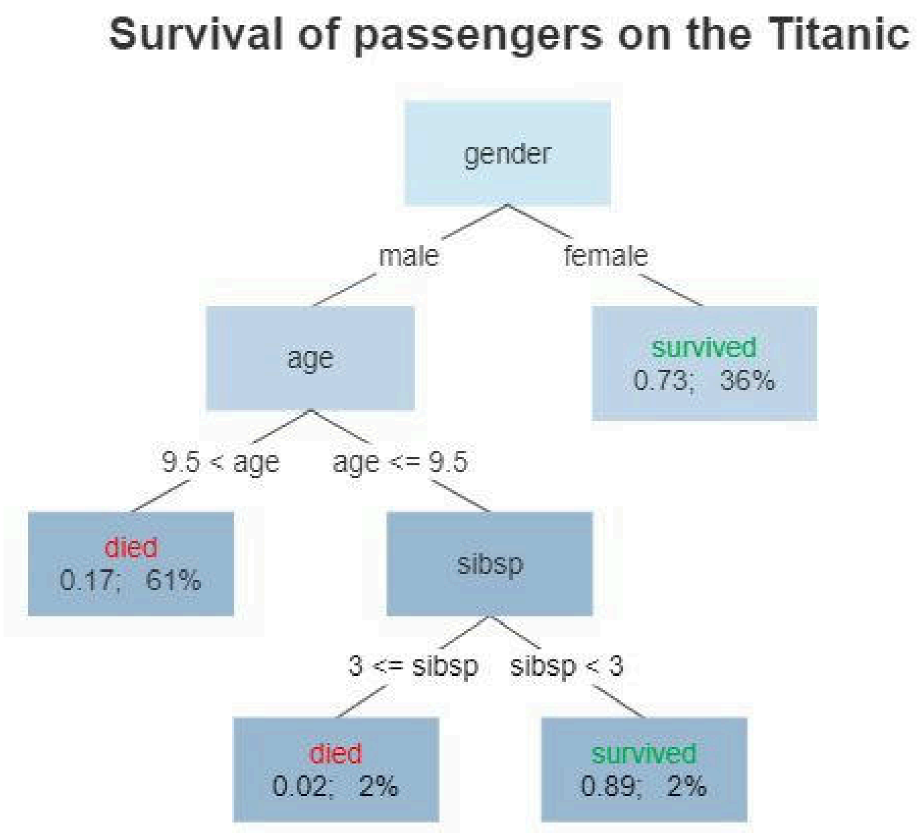
We can get the best of both worlds by training multiple models.

If we split our 100 samples into groups of 20, called *folds*, we can train 5 models. Each model is trained on the other 80 samples and then predicts for a single fold. At the end, we have predictions for each sample, none of which were tainted by the model being able to train on the correct answer.

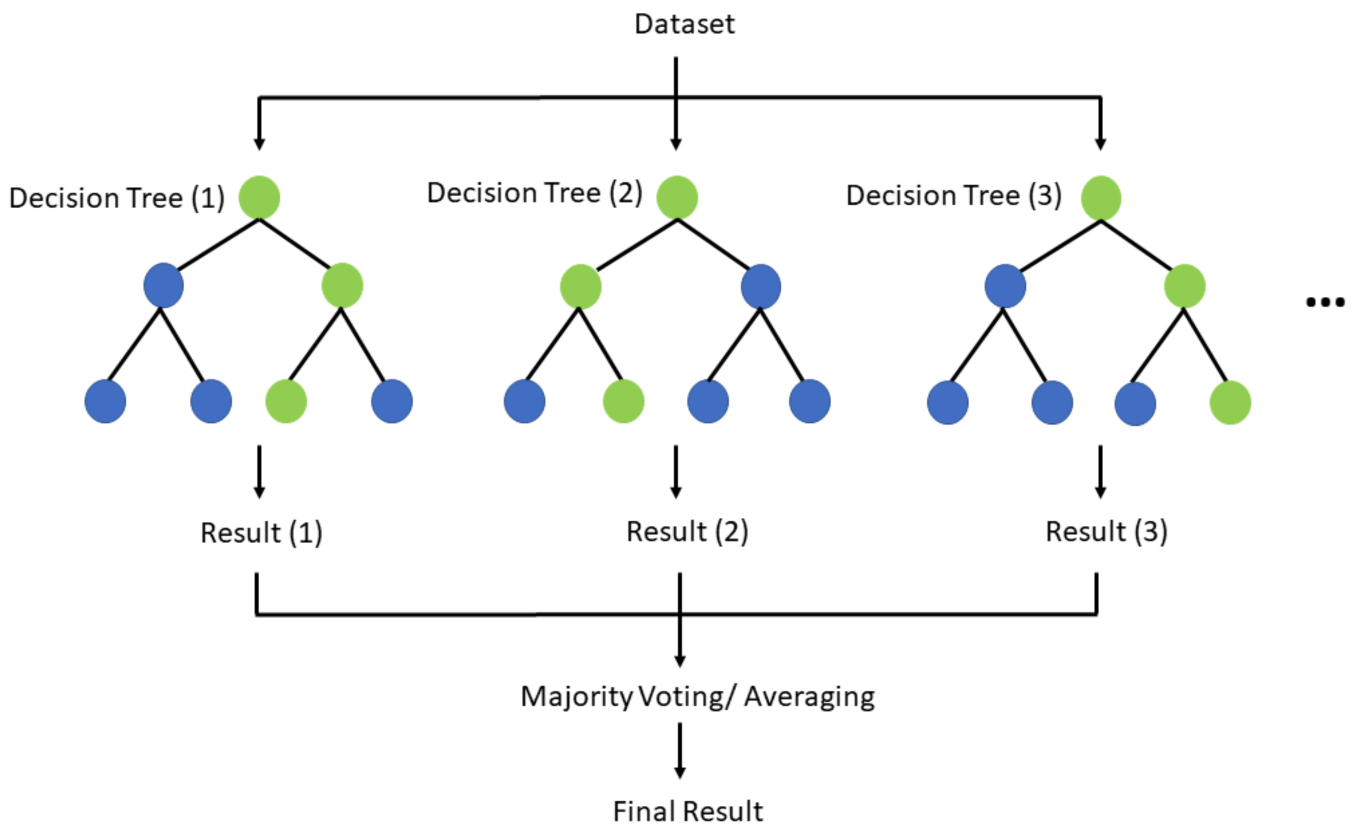
Now our tradeoff is purely computing power. The ideal is *leave-one-out cross-validation* (LOO-CV), where we train a model for every data point. LOO-CV is the best estimate of future performance, but it requires orders of magnitude more computing power. Normally, we use CV with 5 or 10 folds as a more reasonable baseline.

Outline

Decision Tree



Random Forest



Random Forest

Outline

Outline

Gradient Descent

Let's imagine fitting an equation like $ax^2 + bx + c$ to some data.

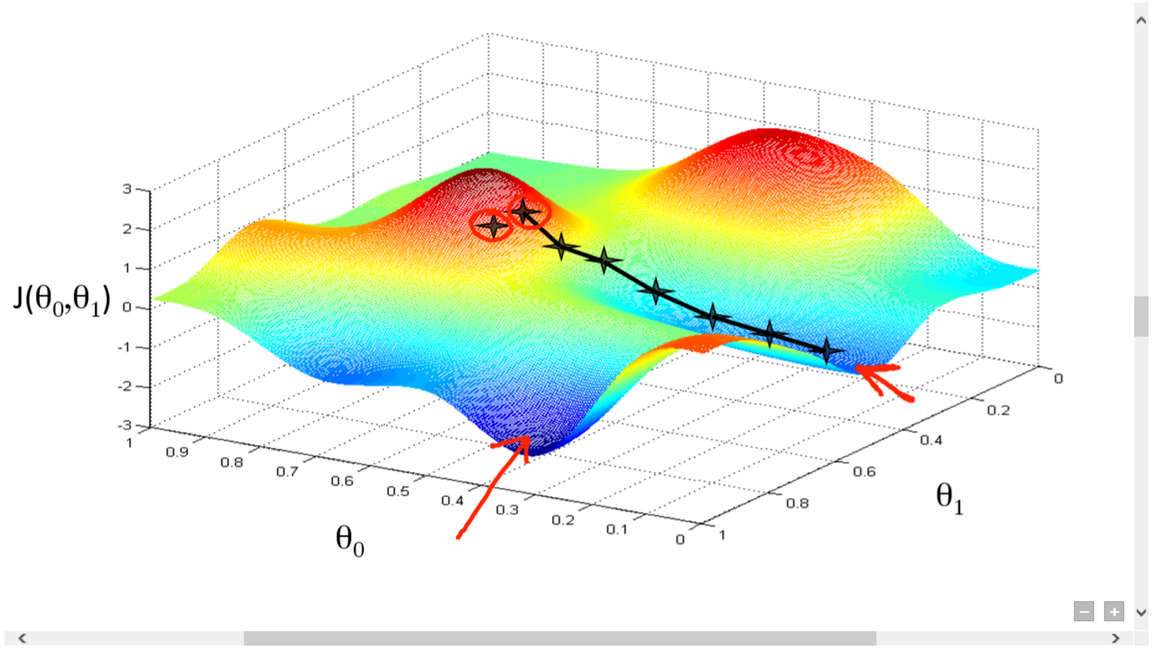
We can define a *loss function* that quantifies the error of our model.

Then, using calculus, we can figure out how changing each parameter a little bit will affect the loss. Using that information, we can change the parameters to decrease the loss. Repeating this enough times will let us find good values of parameters, hopefully.

(Calculating gradients is what PyTorch does, so we won't have to worry about any calculus!)

Gradient Descent

Imagine trying to get down from a mountain in a blizzard, so you can't see at all in front of you. You would probably just try to go downhill, and that's what we do here. We aren't guaranteed to find the best model, but this works well in practice. (Interestingly, this is not at all how humans learn!)



Everything's a Number If You Try Hard Enough

What we just described only works when we have a smooth function that maps numbers to numbers and a quantitative loss function.

How can we make a function that takes in non-numeric input, like a chemical element? A simple method is *one-hot encoding*: we have columns for each element that are 1 or 0 depending on whether the input is that element or not.

$$\begin{aligned} f(H) &= (1, 0, ..., 0) \\ f(\mathrm{He}) &= (0, 1, ..., 0) \\ &\vdots \end{aligned}$$

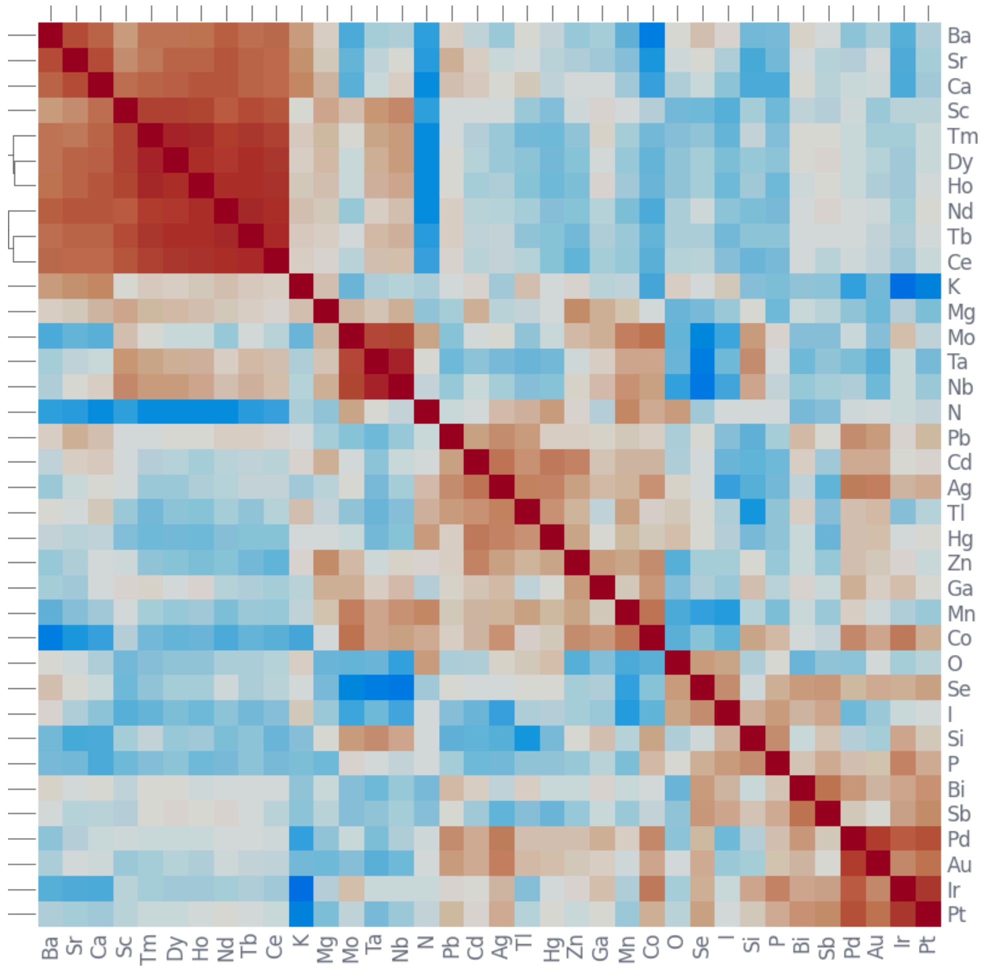
Embeddings

A solution that often works better is to learn a list of numbers (a *vector*) for each potential input. This lets us model how different inputs might be similar.

$$\begin{aligned} f(H) &= (1.2, 4.3, ..., -2.3) \\ f(\mathrm{He}) &= (1.3, -3.1, ..., 0.3) \\ &\vdots \end{aligned}$$

In a language model, maybe the embeddings for *green* and *verde* are similar, so the model can apply what it learns in one language for another language.

Embeddings


Ba
Sr
Ca
Sc
Tm
Dy
Ho
Nd
Tb
Ce
K
Mg
Mo
Ta
Nb
N
Pb
Cd
Ag
Tl
Hg
Zn
Ga
Mn
Co
O
Se
I
Si
P
Bi
Sb
Pd
Au
Ir
Pt

Prelude: Lines

We want a flexible kind of model that can represent all kinds of functions.

For computational reasons, we want this to be built out of simple building blocks.

What’s a simple example of a relationship between some x and y? A line!

$$y = ax + b$$

Composing Neurons

What amazing things can this model do? I’m so excited to find out!

$$\begin{tikzcd}
 & n_1 = ax + b \arrow[dr, "g"] & \\
x \arrow[ur, "a"] \arrow[r, "c"] \arrow[dr, "e"] & n_2 = cx + d \arrow[r, "h"] & gn_1 + hn_2 + in_3 + j \\
 & n_3 = ex + f \arrow[ur, "i"] &
\end{tikzcd}$$

Composing Neurons

This is still just a line. We just made an overly complicated line.

Beyond Linearity

To prevent this network from just being a fancy line, we need to use some function that isn't just multiplication or addition. Basically any function will do.

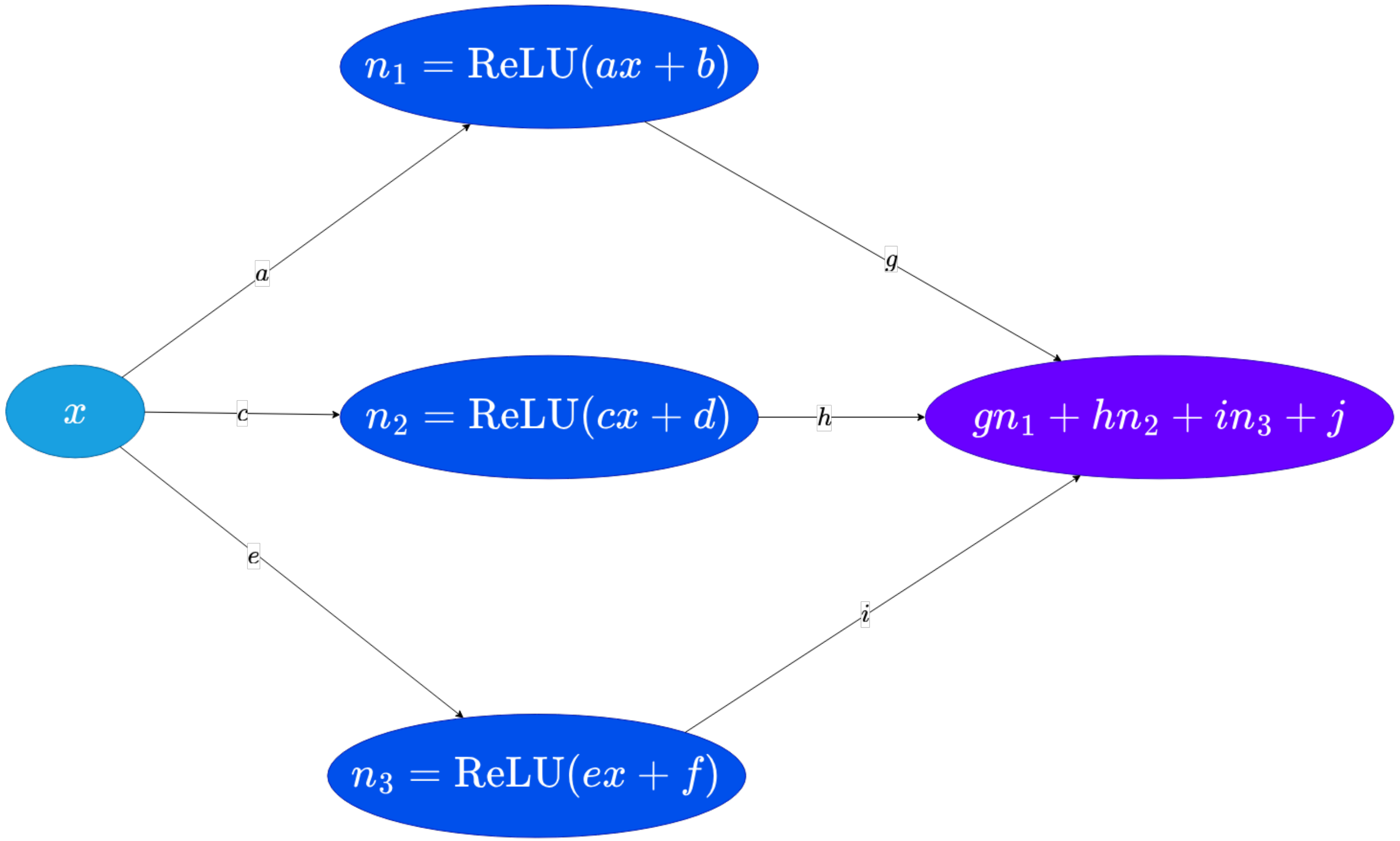
This is called an *activation function*, and people have tried a **lot** of them. Let's use a simple one: the *rectified linear unit*, or ReLU:

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

Above 0, this just returns the input, but below 0 it returns 0.

Let's add this function to the inner neurons:

Beyond Linearity


$n_1 = \mathrm{ReLU}(ax + b)$
x
$n_2 = \mathrm{ReLU}(cx + d)$
$gn_1 + hn_2 + in_3 + j$
$n_3 = \mathrm{ReLU}(ex + f)$
a
c
e
g
h
i

Example

With enough neurons, these networks are extremely flexible. In fact, you can show that they can approximate *any* smooth function with enough neurons.

https://www.desmos.com/calculator/jtcieaiwa7

Outline

Specialized Neural Networks

In theory, with enough neurons even a simple network will work.

In practice, some ways of connecting neurons are better than others for specific tasks. The work of an AI researcher is often trying to find good ways of building networks for a specific task.

Remember: no matter how complex diagrams get, it's just a big equation with sliders. The training process is the same.

Convolutional Neural Network

Neurons respond in the same way to a pattern regardless of where it appears in an image.

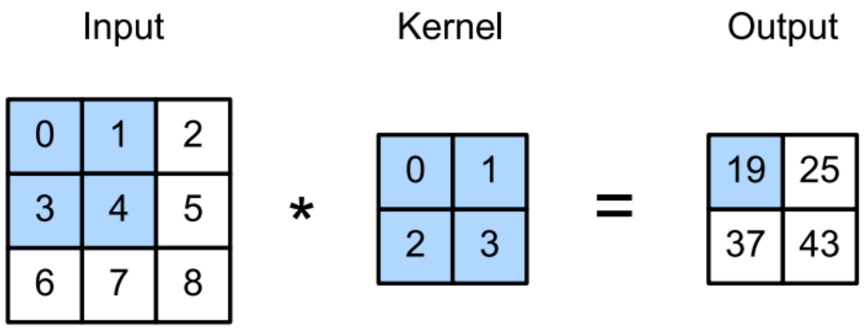


Figure 14.5: Illustration of 2d cross correlation. Generated by code.probml.ai/book1/14.5. Adapted from Figure 6.2.1 of [Zha+20].

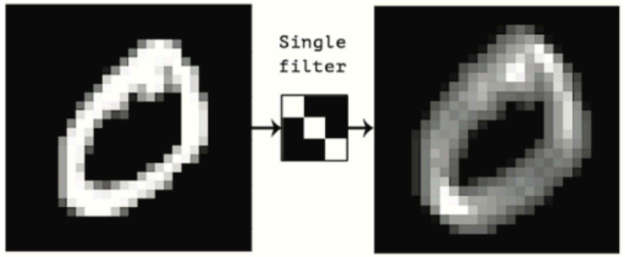


Figure 14.6: Convolving a 2d image (left) with a 3×3 filter (middle) produces a 2d response map (right). The bright spots of the response map correspond to locations in the image which contain diagonal lines sloping down and to the right. From Figure 5.3 of [Cho17]. Used with kind permission of Francois Chollet.

Transformer

The T in ChatGPT, so you might have heard of this one!

In a sequence, compares each element to the others and uses a weighted average of the other values depending on how much each element matches. A neural network is used to do the comparison.

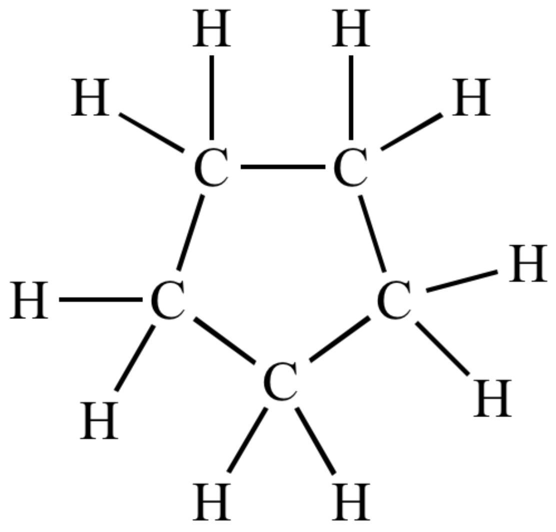
This is called *attention*, and it's very useful for certain kinds of tasks. Here, pronoun shouldn't change depending on the activity: ideally, it would just be estimated from the name.

Nicholas slipped spectacularly on the ice. **He** faceplanted into the earth.

Graph Neural Network

When we say *graph*, we mean a network of nodes and edges. I'll refer to the chart kind of graph as a *plot.*

In a graph neural network, we define functions using neural networks that control how information is communicated between nodes. In the basic formulation, each node sends a *message* to its neighbors and then each node is updated using those messages.



Graph Neural Network

https://distill.pub/2021/gnn-intro/

Outline