Building a Robot Judge: Data Science for the Law

8. Deep Learning Essentials

Elliott Ash

“Neural Networks”

- “Neural”:
  - nothing like brains
- “Networks”:
  - nothing to do with “networks” as normally understood – in particular, nothing to do with network theory in math or social science.

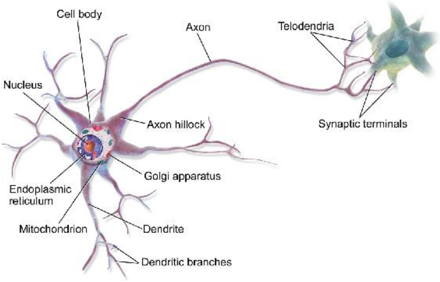
Recent History

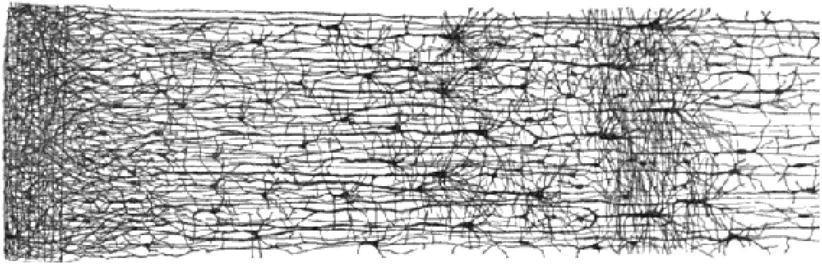
- NNs frequently outperform other ML techniques on very large and complex problems.
- Increase in computing power makes them computationally tractable, graphical processing units (GPUs, designed for video games) give you over 100x performance gain over CPUs.
- Training algorithms have improved – small tweaks have made a huge impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice – for example, they work well on non-convex functions.

Will it last?

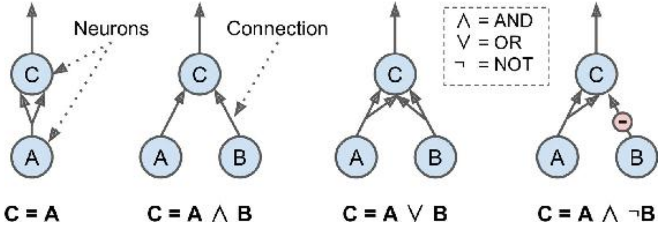
- Three key principles of deep learning will persist:
  - **Simplicity**
    - feature engineering is obsolete
    - complex, brittle, engineering-heavy pipelines replaced with simple, end-to-end trainable models, composed of 5-6 tensor operations.
  - **Scalability**
    - amenable to parallelization on GPUs or TPUs (tensor processing units)
    - trained on batches of data, so can be scaled to datasets of arbitrary size.
  - **Versatility and reusability**
    - can be trained on additional data without restarting from scratch, therefore amenable for continuous online learning.
    - deep-learning models are repurposable and thus reusable

Biological Neurons



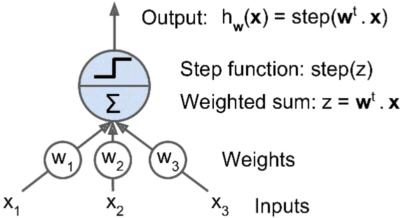


Neurons can perform logical computations



- These networks perform the identity, logical AND, logical OR, and logical NOT operations.

Perceptron LTU



- In a perceptron, an individual neuron (called an LTU, or linear threshold unit) is defined by

$$h(\boldsymbol{x}) = \text{step}(\boldsymbol{\omega}'\boldsymbol{x})$$

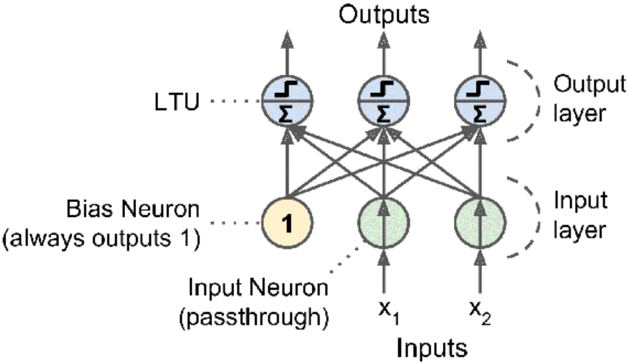
where

$$\text{step}(a) = \begin{cases} 0 & a < 0 \\ 1 & a \geq 0 \end{cases}$$

  - The neuron computes a linear combination of the inputs; if result exceeds threshold, output positive class, otherwise negative class.

Perceptron

- A perceptron is an array of LTUs in parallel:



- This basic perceptron is similar to a logistic regression model.

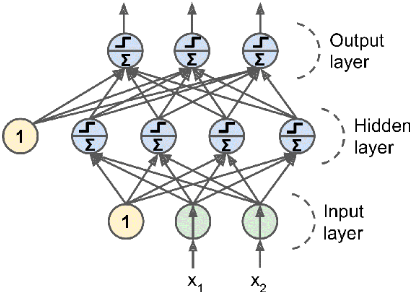
In Notation

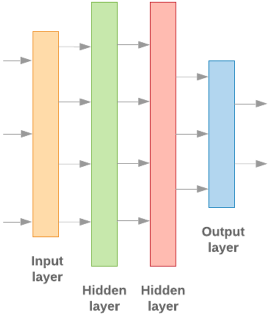
- The simplest perceptron is a linear combination of the inputs, the same as a linear regression:

  $$y = \alpha + x'\omega$$

  where x is a vector of inputs and ω is the vector of weights (or a matrix for a multi-class outcome).

- The predictive performance of perceptrons improved substantially by stacking them into multiple layers:





- Input variables are connected to multiple neurons in the hidden layer(s), which in turn are connected to output layer.
  - This is called a multi-layer perceptron or a feed forward neural network; with enough neurons, it can approximate any continuous function.
  - This is the "deep" in deep learning!

DNN Notation

- A DNN with a single hidden layer can be written as

$$y = \alpha_2 + g(\alpha_1 + x'\omega_1)'\omega_2$$

  - α_1 and ω_1, the intercept and coefficients in the input layer
    - ω_1 is a $d_0 \times d_1$ matrix, where d_0 is the dimension of the input and d_1 is the number of neurons in the hidden layer.
  - $g(\cdot)$, the non-linear activation function.
    - without this, the DNN could only represent linear transformations of the input.
  - α_2 and ω_2, the intercept and coefficients in the hidden layer.
    - ω_2 is a $d_1 \times d_2$ matrix, where d_2 is the dimensionality of the output.

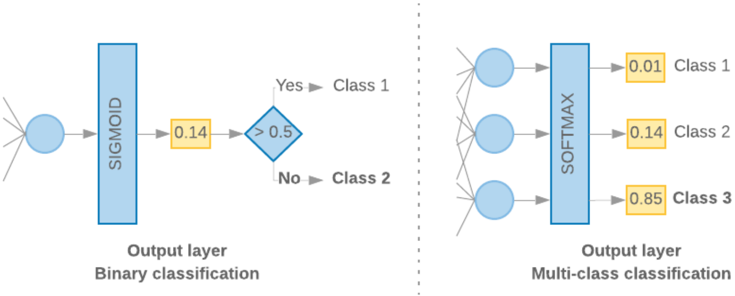
DNN Notation: two hidden layers

- Similarly, with two hidden layers we have

$$y = \alpha_3 + g_2(\alpha_2 + g_1(\alpha_1 + x'\omega_1)'\omega_2)'\omega_3$$

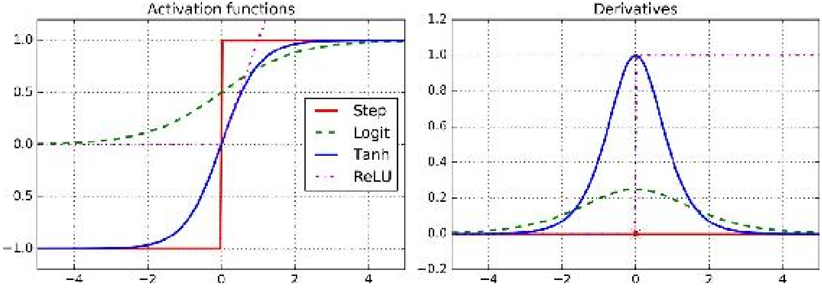
  - $g_1(\cdot)$ and $g_2(\cdot)$, activation functions for the first and second layers.
  - α_3 and ω_3, intercepts and coefficients for the second hidden layer.

Constructing the Last Layer



- MLPs will output a probability distribution across output classes.
  - can also output a real number, which would make a regression model.

Modern MLPs: New activation functions



- logistic function: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  - ranges between -1 and 1 (rather than between 0 and 1, as the case with the logistic)
  - centered on zero, can speed up convergence
- ReLU (rectified linear unit) function: $\max\{0, z\}$,
  - deceptively simple, fast to compute, and very effective in practice
  - gradient does not saturate to zero for large values (but is flat below zero)

Google Developers Advice: MLP baseline for Text Classification

1. Calculate the number of samples/number of words per sample ratio.
2. If this ratio is less than 1500, tokenize the text as n-grams and use a simple multi-layer perceptron (MLP) model to classify them.

   - In the case of N-grams models, Google testers found that MLPs tended to out-perform logistic regression and gradient boosting machines.

Python Implementation

- See the Jupyter notebook for Keras examples.
  - has not been updated to Keras 2.0 yet.
- "Dense" layer is the DNN baseline – means that all neurons are connected.
- Output layer:
  - for binary classification, use `activation='sigmoid'`
  - for regression, do not use an activation function
  - for multi-class classification, use `activation=softmax'`

Loss function and metrics

- Loss function:
  - for binary classification, use `binary_crossentropy`
  - for regression, use `mean_squared_error`
  - for multi-class classification, use `sparse_categorical_crossentropy`
- Metrics:
  - for classification, can use accuracy
  - for regression, can define a custom metric (see accompanying code)

Tuning NN Hyperparameters

- Number of hidden layers:
  - having a single hidden layer will generally give decent results.
    - more layers with fewer neurons can recover hierarchical relations and complex functions
    - for text classification, try one or two hidden layers as a baseline.
- Number of neurons:
  - a common practice is to set neuron counts like a funnel, with fewer and fewer neurons at each level
  - or just pick 150 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Activation functions:
  - ReLU works well for hidden layers
  - softmax is good for the output layer in classification tasks

Xavier and He Initialization

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Connection weights should be initialized randomly according to a uniform distribution or normal distribution, as indicated in the table (see Geron Chapter 11).

Other Activation Functions

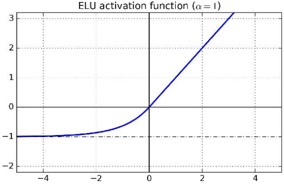
- Leaky ReLU

$$\max(\alpha z, z)$$

where α is set to a small number, such as .01, or learned in training.

- Exponential linear unit

$$\text{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



- In general, ELU has had the best performance so far, but it is slower than ReLU.

Batch normalization

- Another trick to speed up training:
  - in between layers, zero-center and normalize the inputs to variance one.
  - normally done before a non-linear activation function

Regularization for Sparse Models

- As with linear models, neural network parameters can be regularized with an L1 and/or L2 penalty to push weak neurons to zero and produce a sparse model.

Dropout

- An elegant regularization technique:
  - at every training step, every neuron has some probability (typically 0.5) of being temporarily dropped out, so that it will be ignored at this step.
  - after training, neurons dont get dropped any more.
- Neurons trained with dropout:
  - cannot co-adapt with neighboring neurons and must be independently useful.
  - cannot rely excessively on just a few input neurons; they have to pay attention to all input neurons.
    - makes the model less sensitive to slight changes in the inputs.
- If a model is over-fitting, increase dropout. Dropout can be higher for large layers and lower for small layers.

Optimizers

- Choice of optimization algorithm is the topic of active research, which has shown that it can have a big impact on model performance.
  - Until recently, a good starting choice would be Adam (adaptive moment estimation), which is fast and usually works well. For robustness, can also try SGD.
  - A recent paper says that AdaBound dominates Adam or SGD.

Early stopping

- A popular/efficient regularization method is to continually evaluate your model at regular intervals, and then to stop training when the test-set accuracy starts to decrease.

Practical Guidelines

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Source: Geron book.

Batch Training with Large Data

- If data sets don't fit in memory, one can load the data in batches from disk.
- can also continuously update a saved model.

Grid search for model choice

- The flexibility of DNNs is a blessing and a curse.
  - in general, one should make a complex model that allows regularization.
- But still, there are many choices to be made.
  - to choose the number of hidden layers, for example, one can use cross-validation grid search (as we did with standard scikit-learn models).