"From 2016 to 2020, the entire machine learning and data science industry has been dominated by two approaches: deep learning and gradient boosted trees. Specifically, gradient boosted trees is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification. . . . These are the two techniques you should be most familiar with in order to be successful in applied machine learning today"

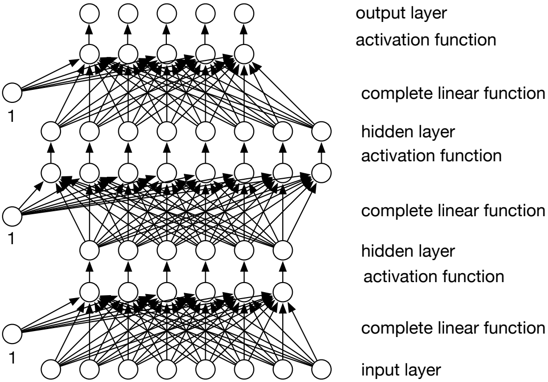
F. Chollet [2021]

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Y. LeCun, Y. Bengio, G. Hinton [2015]

Feed-forward neural networks

- Feed-forward neural networks are directed acyclic graphs:



- Each hidden unit outputs a linear function of its inputs followed by a non-linear activation function.

Neural network properties

- The depth of a neural network is the number of layers.
- The width of a layer is the number of elements in the vector output of the layer.
- The width of a neural network is the maximum width over all layers.
- The size of the output and input are usually specified as part of the problem definition.

Feed-forward neural networks

- A feed-forward neural network implements the function

$$f(x) = f_n(f_{n-1}(\dots f_2(f_1(x))))$$

- x is a vector of input values (the input layer)
- Each function f_i maps a vector into a vector.
- Each component of an output vector is called a unit.
- Function f_i is the ith layer.
- The last layer, f_n, is the output layer.
- The other layers are called hidden layers.
- The number of functions, n, is the depth of the network.
- "Deep" in deep learning refers to the depth of the network.

Feed-forward neural networks

Each layer f_i is

- a linear function with learnable parameters of each output given the input
- followed by a non-linear activation function, ϕ.
- The linear function takes a vector *in* and an extra constant input with value "1", and returns a vector *out*:

$$out[j] = \phi(\sum_k in[k] * w[k, j])$$

  for a 2-dimensional array w of weights.
- The weight associated with the extra 1 input is the bias.
- The weight $w[i, j]$ for each input–output pair of the layer, plus a bias for each output.
- The outputs of one layer are the inputs to the next.
-

Neural network inputs

- The input to a neural network is a vector of real numbers.
- Boolean variables are represented using 1 for true and either 0 or -1 for false.
- Categorical variables can be represented using indicator variables – a binary variable for each value – forming a one-hot encoding

Activation function: ReLU

- A common activation function is the rectified linear unit (ReLU):

$$phi(x) = max(0, x)$$

or

$$\phi(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

- The derivative of ϕ is

$$\frac{\partial \phi}{\partial x}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Activation function for output

The activation function and what is being optimized depends on the type of the output layer:

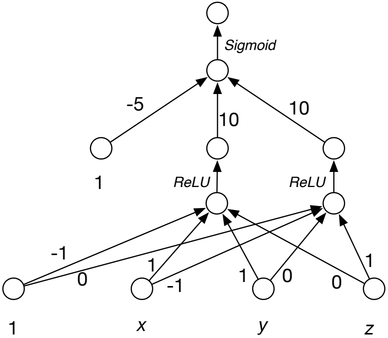
- If output is real, optimize squared loss, and use the identity function: $\phi(x) = x$
- If output is Boolean, use binary log loss, with a sigmoid function:

$$\phi(x) = sigmoid(x) = \frac{1}{1 + \exp(-x)}$$

- If output y is categorical, but not binary, use categorical log loss with a softmax function. The output layer has one unit for each value in the domain of y (for example, MNIST).

What can a neural network represent?

The function "if x then y else z" cannot be represented using logistic regression. It can be approximated with the neural network:



The function can be represented as $(x \wedge y) \vee (\neg x \wedge z)$

Gradient Descent

- If the domains are continuous, Gradient descent movies each each variable downhill; proportional to the gradient of the heuristic function in that direction.
  The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$.
  η is the step size.
- Neural networks do gradient descent with many parameters (variables) to minimize an error on a dataset. Some large language models have over 10^{12} parameters.

Differentiation

Two properties of differentiation are used in backpropagation:

- Linear rule: the derivative of a linear function, $aw + b$, is given by:

$$\frac{\partial}{\partial w}(aw + b) = a$$

- Chain rule: if g is a function of w and function f, which does not depend on w, is applied to $g(w)$, then

$$\frac{\partial}{\partial w} f(g(w)) = f'(g(w)) * \frac{\partial}{\partial w} g(w)$$

where f' is the derivative of f.

Use of chain rule

A network represents $f(e) = f_n(f_{n-1}(\dots f_2(f_1(x_e))))$, where example e has features x_e. Suppose $v_i = f_i(v_{i-1})$ and $v_0 = x_e$. Consider weight w used in the definition of f_j:

$$\begin{aligned}
&\frac{\partial}{\partial w} error(f(e)) \\
&\quad = error'(v_n) * \frac{\partial}{\partial w} f_n(v_{n-1}) \\
&\quad = error'(v_n) * \frac{\partial}{\partial w} f_n(f_{n-1}(v_{n-2})) \\
&\quad = error'(v_n) * f_n'(v_{n-1}) * \frac{\partial}{\partial w} (f_{n-1}(v_{n-2})) \\
&\quad = error'(v_n) * f_n'(v_{n-1}) * f_{n-1}'(v_{n-2}) * \dots * \frac{\partial}{\partial w} (f_j(v_{j-1}))
\end{aligned}$$

where f_i' is the derivative of f_i with respect to its inputs.

Backpropagation

- Backpropagation implements (stochastic) gradient descent for all weights.
- Two passes:
  - Prediction: given inputs compute outputs of each layer
  - Back propagate: Going backwards,

    $$error'(v_n) * \prod_{i=0}^{k} f'_{n-i}(v_{n-i-1})$$

    for k starting from 0 are computed and passed to the lower layers. Weights in each layer are updated.

class Dense(n_i, n_o) ▷ n_i is # inputs, n_o is #outputs

    for each $0 \le i \le n_i$ and each $0 \le j < n_o$ **do**

        $d[i, j] := 0$; $w[i, j] :=$ a random value

    def *output*(in) ▷ *in* is array with length n_i

        for each j **do** $out[j] := w[n_i, j] + \sum_i in[i] * w[i, j]$

        return *out*

    def *Backprop*($error$) ▷ *error* is array with length n_o

        for each i, j **do** $d[i, j] := d[i, j] + in[i] * error[j]$

        for each i **do** $ierror[i] := \sum_j w[i, j] * error[j]$

        return *ierror*

    def *update*() ▷ update weights. η is learning rate.

        for each i, j **do**

            $w[i, j] := w[i, j] - \eta / batch\_size * d[i, j]$

            $d[i, j] := 0$

Neural-network learner

functions is the list of functions that compose the neural network.

repeat
  batch := random sample of *batch_size* examples
  for each example e in *batch* **do**
    for each input unit i **do** $values[i] := X_i(e)$
    for each *fun* in *functions* from lowest to highest **do**
      $values := fun.output(values)$
    for each output unit j **do**
$error[j] := \phi_o(values[j]) - Ys[j]$
    for each *fun* in *functions* from highest to lowest **do**
      $error := fun.Backprop(error)$
  for each *fun* in *functions* that contains weights **do**
    $fun.update()$
until termination

RMS-Prop Optimization

- In RMS-Prop the magnitude of the change in a weight depends on how its gradient compares to its historic value.
- It maintains r, a rolling average of the square of the gradient.
- For a dense layer, the update method becomes:

```
1: def update()                                    ▷ update weights
2:     for each i, j do
3:         g := d[i,j]/batch_size
4:         r[i,j] := ρ * r[i,j] + (1 − ρ) * g^2
5:         w[i,j] := w[i,j] − (η * g) / sqrt(r[i,j] + ε)
6:         d[i,j] := 0 .
```

- ϵ ($\approx 10^{-7}$) is used to ensure numerical stability.
- When $r[i,j] \approx g^2 \gg \epsilon$, the ratio $g/\sqrt{r[i,j]+\epsilon}$ is approximately 1 or -1, depending on the sign of g.
- When $g^2 \ll r[i,j]$, the step size is smaller than η.

Initialization

- Real-valued variables are normalized by subtracting the mean, and dividing by the standard deviation.
- In a one-hot encoding, categorical input variable X with domain $\{v_1, \ldots, v_k\}$ is represented as k input indicator variables, $X_1, \ldots, X_k$. An input example with $X = v_j$ is represented with $X_j = 1$ and every other $X_{j'} = 0$.
- What happens if the weights in the hidden layers are all set to the same value?
- For the output units, non-bias weights can be set to zero and the bias weights to the mean for regression or inverse-sigmoid of the empirical probability for classification. (Why?)

Pragmatics of Training Neural Networks

- Make sure it is learning something: The error on the training set should beat a naive baseline corresponding to the loss being evaluated.
- If the performance on the training set is poor, change the model.
  (Poor performance on the training set indicates under-fitting.)
- Test the error on the validation set. If the validation error does not improve as the algorithm proceeds, it means the learning is not generalizing, and it is fitting to noise.
  (Poor performance on the validation set indicates overfitting.)
  In this case you should simplify the model.
- Then carry out hyperparameter tuning.
-
-

Hyperparameter Tuning

The hyperparameters that can be tuned include:

- the algorithm (a decision tree or gradient-boosted trees may be more appropriate than a neural network)
- number of layers
- width of each layer
- number of epochs, to allow for early stopping
- learning rate
- batch size
-