Neural Networks

Part 2

11 November 2020
Where are we?
Learning
People have a remarkable capacity for pattern recognition.

Our brains process inputs from the world – vision, hearing, touch, taste – and categorize them – a chair, “hello”, soft, sweet – so that we can act appropriately.

We do this automatically and quickly, with little effort.
We recognize these digits without consciously thinking “that looks like two orthogonal lines connected to a rotated semi-circle, so that’s a 5”.

504192
Designing rules to recognize handwritten digits is *complicated*, which made it hard to program computers to do so.

Neural networks let us solve problems like this by loosely mimicking the way our brains solve them: Taking inputs, processing them, and generating an output, using patterns learned from exposure to examples.
DICK TRACY

YOUR BROTHER WAS WORKING ON A "NEURAL-NETWORK" COMPUTER? YOU MEAN, ARTIFICIAL INTELLIGENCE?

AH—YOU DO KEEP UP, DON'T YOU, MR. TRACY?

"YES—A COMPUTER THAT CAN LEARN THROUGH EXPERIENCE—"
Perceptrons
Perceptrons were developed in the 1950s. Today we use different models of artificial neurons, but they follow the general design of the perceptron.
A perceptron combines one or more input values to output 0 or 1.

If the inputs are the outputs of other perceptrons, then they’ll be binary too.
Weather nice?

Weekday?

Walk to class?
If it’s a sunny weekday, $1 \times 2 + 1 \times 6 = 8$

So, do we walk to class?
We need a threshold, e.g.,

\[ f(x) = 1 \text{ if } x \geq 5; \text{ otherwise } 0 \]
output = \begin{cases} 
0 & \text{if } \sum_i w_i x_i < \text{threshold} \\
1 & \text{if } \sum_i w_i x_i \geq \text{threshold} 
\end{cases}

Training the perceptron is easier if we move the threshold to the other side of the inequality and replace it with the neuron’s bias:

output = \begin{cases} 
0 & \text{if } \sum_i w_i x_i + \text{bias} < 0 \\
1 & \text{if } \sum_i w_i x_i + \text{bias} \geq 0 
\end{cases}

Note that bias = −threshold
Why did we do this?

Just as we don’t know the right weights before training the perceptron, we don’t know the right threshold.

Making it a bias term just makes it another weight that we train, albeit one that we multiply to a fixed “input” of 1.
Thus we can also rewrite this sum-of-products plus the bias as a dot product of two vectors:

\[
\text{output} = \begin{cases} 
0 & \text{if } \sum_i w_i x_i + \text{bias} < 0 \\
1 & \text{if } \sum_i w_i x_i + \text{bias} \geq 0
\end{cases}
\]

Thus we can also rewrite this sum-of-products plus the bias as a dot product of two vectors:

\[
\text{output} = \begin{cases} 
0 & \text{if } \mathbf{w} \cdot \mathbf{x} < 0 \\
1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0
\end{cases}
\]
Perceptrons represent a decision surface:
Perceptrons can only classify linearly separable data.
Learn perceptron weights from data

**Key idea:** Loop through the training data and update weights when wrong.

initialize weights (to 0 or random values)
while any training example is misclassified:
    for each training input $i$:
        $\text{output} = \text{compute_perceptron_output}(\text{input})$
        $\text{correction} = \text{target}[i] - \text{output}$
        for each input weight $w[i]$:
            $w[i] += \text{learning_rate} \times \text{correction} \times \text{input}[i]$
If we arrange multiple perceptrons in a single layer, we can have multiple possible outputs, such as identifying which of the digits 0–9 an image depicts.
Neural networks
A *neural network* is a graph where neurons (units) are organized in layers.

Typically, consecutive layers are fully connected.
One or more perceptrons constitute a neural network:

Topology:
- Input layer
- Output layer

All activation functions are thresholds

One input is a constant 1
- Allowing for a bias while fixing the threshold at 0
Other models of neurons we can use in our neural networks follow the basic design of the perceptron, but they vary in the choice of activation function that’s applied to the weighted sum before output.
Last time, we made our network a little more complex, with one or more **hidden layers** between the input layer and the output layer:
Last time, we made our network a little more complex, with one or more *hidden layers* between the input layer and the output layer:
This type of network is called a *feedforward neural network* because the information flows in one general direction: forward from the input layer to the output layer.

When there’s more than one hidden layer, it’s called a *deep neural network*.
The reason hidden layers are so important is that they can *discover features* in the data and allow the following layers to operate on those features rather than on the (noisy, large) raw inputs.
THEOREM (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.
Parameters:

- Network topology
  - Number of hidden layers
  - Size of each hidden layer
  - Connectivity of hidden layers
- Activation functions
- Edge weights

Often hand-picked (though they can be learned experimentally)

Learned
How do we measure how well our neural network works?

For a set of data – e.g., our training set – we compute the average error over all the examples.

For each example, we typically measure the mean square error – that is, the difference between the actual value and the predicted value, squared: \((\text{target} - \text{output})^2\)

If we use this error calculation as a guide for training our model, we call it the loss function (or cost function).
We want to train the network to minimize the loss.

However, the simple learning rule used for perceptrons doesn’t work if you have multiple layers.

When we encounter an error in the output, we adjusted the the weights of the perceptron in the output layer.

But a *small change* in the weights or bias for any perceptron in the network could cause its output to *flip* (0→1 or 1→0). If there are multiple layers, this can make the rest of the network change in big ways!

It’s not clear how we should adjust the weights of perceptrons in the previous layers to minimize the loss.
Researchers eventually found an answer, using calculus.

But to make it work, they needed to change the model of neurons.
The output of a perceptron (0 or 1) is determined by its activation function, the *Heaviside* step function:

\[
f(x) = \begin{cases} 
0 & \text{if } x < 0 \\
1 & \text{if } x \geq 0 
\end{cases}
\]
Other activation functions don’t force the output to be just 0 or 1. For example, the sigmoid (or logistic) function:

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

These continuous values can be considered the probability the output should be a “yes”.
Crucially, our choice of activation function is still **non-linear**.

If we switched to a linear function, we’d only be able to model linear relations, which would make neural networks much less powerful!

But because it’s continuous, it now has the advantage of being differentiable.
So, we can find how much the error changes if we change any weight in the neural network, including those in the hidden layers.

We use an optimization technique – typically *stochastic gradient descent* – to find the optimal weights to minimize the error.

In stochastic gradient descent, we update the error from one input at a time, in random order.

Because error is many-peaked, stochasticity (randomness) is a good thing – it helps escape bad local minima.

This is similar to random moves in hill-climbing (local search).
The chain rule is used to compute the derivative of the error for all the neurons in a prior layer and thus we know the way to adjust their weights to reduce the error.

That is, when the output layer makes a mistake, we can use calculus to assign some of the blame to each neuron in the previous, hidden layer.

We keep doing this as long as there's another hidden layer.

This is called backpropagating.
Training

Inference

Backpropagation is the key algorithm that makes training multilayer neural models computationally tractable.

It can make training with gradient descent as much as ten million times faster, relative to a naive implementation.

*That said, it’s ok (for this class) if you treat backpropagation as a black box.*
Convolutional neural networks
“Classical work in visual pattern recognition has demonstrated the advantage of extracting *local features* and *combining them to form higher order features*. Such knowledge can be easily built into the [neural] network by forcing the hidden units to combine only local sources of information. Distinctive features of an object can appear at various location on the input image. Therefore it seems judicious to have a set of feature detectors that can detect a particular instance of a feature *anywhere* on the input place.”

The basic neural network architecture we’ve seen isn’t always the most appropriate.

Instead, we can have the early hidden layers consist of identical units with identical input weights, each taking inputs from a small local “patch” of the input (where patches slightly overlap).

Computing the weighted sum of the elements of each input patch, performed over identical, slightly overlapping patches that “tile” (cover) the entire input pattern, amounts to mathematical convolution.
Instead of each neuron having a different weight for each pixel of the input image, the neurons only have a small set of weights that were applied to many small subsets of the image.

Later layers work in a similar way, but acting on the “local” features found in the previous hidden layer rather than pixel images, so they “see” successively larger portions of the image.

The last two layers are normal neural net layers.
10 output units fully connected

30 hidden units fully connected

12 × 16 hidden units

12 × 64 hidden units

16 × 16 digitised grayscale images
http://yann.lecun.com/exdb/lenet/
Without such constraints, the network would have to learn the same simple things (such as detecting 45° lines, small circles, etc.) many times for each portion of the image.
Since the exact pixel locations of such features don’t matter, the neuron could basically skip neighboring subsets of the image – *subsampling*, now known as a type of *pooling* – when applying the weights, further reducing the training time.
The addition of these two types of layers – convolutional and pooling – are the primary distinctions of *convolutional neural nets* (ConvNets) from standard neural nets.
Convolutional neural networks were important in part because *backpropagation doesn’t work well for normal neural networks with many layers*. Why?

Backpropagation relies on finding the error at the output layer and successively splitting up blame for it to prior layers.

With many layers, this calculus-based splitting of blame ends up with either huge or tiny numbers (the “vanishing or exploding gradient problem”).
Deep learning
“Deep learning” started in 2006 with the paper “A fast learning algorithm for deep belief nets” by Hinton et al.

The important idea was that neural networks with many layers could be trained well – if the weights are initialized in a clever way rather than randomly.
That clever way of initializing weights:

Train each layer, one by one, with unsupervised training (learning a generative model of the data).

This starts off the weights much better than giving them random values.

Then finish with a round of supervised learning, as normal.
“Research on deep architectures is in its infancy, and better learning algorithms for deep architectures remain to be discovered. Taking a larger perspective on the objective of discovering learning principles that can lead to AI has been a guiding perspective of this work.”

Algorithmic advances alone didn’t overcome the early limitations of neural networks.

In addition to an optimization algorithm, you need data!

Difficult tasks require lots of data.
Switching to GPUs: 70 times faster!
The more layers there are, the worse backpropagation works for training.

Why? Glort & Bengio (2010) found that:

The particular non-linear activation function chosen for neurons makes a big impact on performance, and the sigmoid often used is not a good choice.

The “vanishing gradient” problem happens when you choose random weights without consideration for which layer the weights are for.

Backpropagation involves a sequence of multiplications that result in smaller derivatives for earlier layers.

Alternative: Choose random weights with different scales according to the layer they’re in.
So, what’s the best activation function?

It’s ReLU: \( f(x) = \max(0, x) \)

Weird result since it’s not strictly differentiable.

(It’s not differentiable precisely at 0.)

We’re not sure exactly why this is the case, though there have been some suggestions:

It leads to sparse representations (i.e., not many neurons end up needing to be non-0)

The function is simple to compute and thus faster than sigmoid (exponential) or tanh (trigonometric).

The constant (0 or 1) derivative of ReLU helps avoid the vanishing gradient problem.
Why didn’t supervised learning with backpropagation work well in the past?

Per Geoffrey Hinton:

- Labeled data sets were thousands of times too small.
- Computers were millions of times too slow.
- We initialized weights in a stupid way.
- We used the wrong type of non-linearity.
Deep learning =
Lots of training data +
Parallel computation +
Scalable, smart algorithms
Since 2012, deep learning has revolutionized much of AI as a field; this has been the decade of deep learning.
Acknowledgments

The lecture incorporates material from:

Andrey Kurenkov, “A Brief History of Neural Nets and Deep Learning”
Michael Nielson, *Neural Networks and Deep Learning*
Bryce Wiedenbeck, Swarthmore
Ketrina Yim (illustrations)