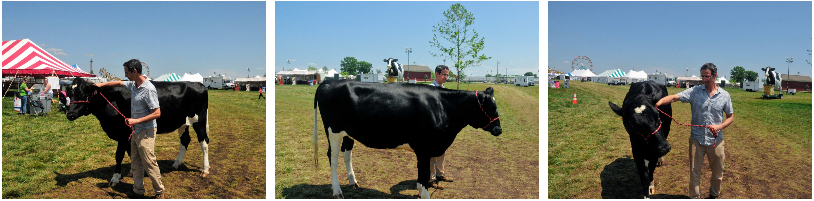
Multilayer Perceptrons

Nan Ye

School of Mathematics and Physics
The University of Queensland

The Journey Begins with a Cow



Guess the weight of the cow: poll

Where Are We Heading to?

How to ~~accurately estimate the cow weight~~ build good ML models

- Making use of a crowd ⇒ Week 7 Ensemble methods

  each of us is a biological prediction model trained on different datasets...

- Using a neural network ⇒ Week 8 and 9 Neural networks

  brain-inspired models, some are good for images...

- Making a robust model ⇒ Week 10 Robust machine learning

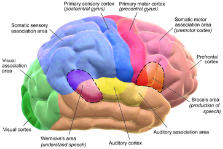
  malicious users, outliers,...

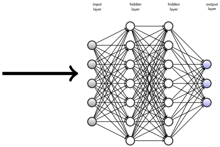
- Asking for explanations ⇒ Week 11 Interpretable machine learning

  Human behavior cannot be explained for the simple reason that it makes no sense. – Martin Rubin
  ...still, let's ask the machines for explanations...

- Exploiting prior beliefs ⇒ Week 12 Bayesian methods

Brain-inspired Machine Learning


Primary sensory cortex (postcentral gyrus)
Primary motor cortex (precentral gyrus)
Somatic sensory association area
Somatic motor association area (premotor cortex)
Visual association area
Prefrontal cortex
Broca's area (production of speech)
Visual cortex
Auditory association area
Wernicke's area (understand speech)
Auditory cortex



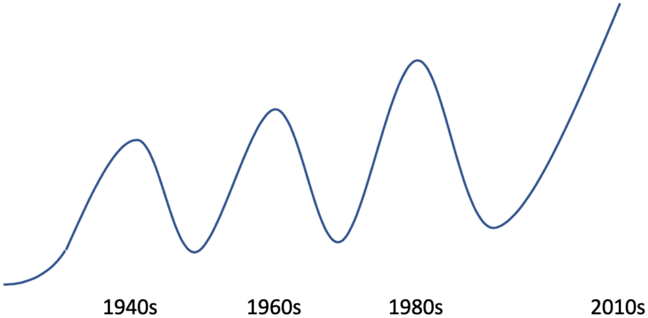
von Neumann compuer
✓ numerical problems
✓ symbolic problems
✗ perceptual problems
✗ high energy
✗ stored programs
✗ not robust

human brain
✗ numerical problems
✗ symbolic problems
✓ perceptual problems
✓ low energy
✓ self-learning
✓ robust

artificial neural net
automatic speech recognition
game AI
face recognition
medical image analysis
document classification
...

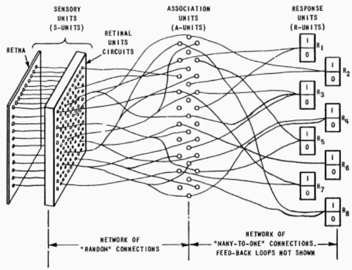
Artificial neural nets (ANNs) draw inspiration from how human brain works and have found many successful applications in recent years

History of ANNs

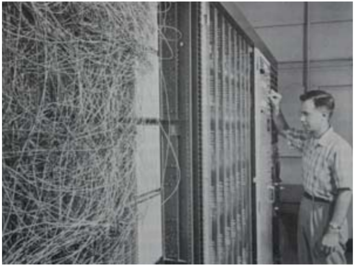


Research on artificial neural networks have gone through several ups and downs since its inception (figure is approximate).

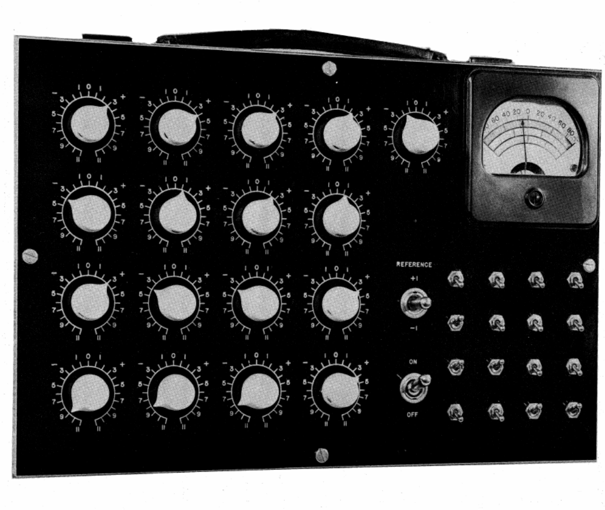
(A Few) Pioneers


SENSORY UNITS (S-UNITS)
RETNA
RETINAL UNITS CIRCUITS
ASSOCIATION UNITS (A-UNITS)
RESPONSE UNITS (R-UNITS)
R_1
R_2
R_3
R_4
R_5
R_6
R_7
R_8
1
0
NETWORK OF "RANDOM" CONNECTIONS
NETWORK OF "MANY-TO-ONE" CONNECTIONS. FEED-BACK LOOPS NOT SHOWN

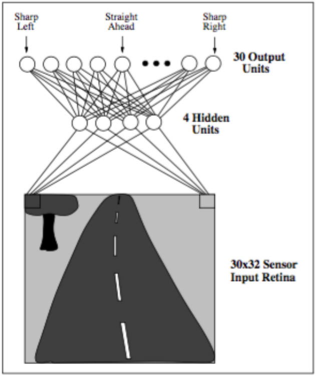
Mark I Perceptron architecture



Rosenblatt and Mark I Perceptron



The Adaline is a lunch box sized machine



ALVINN Driving at 70 MPH (1993)

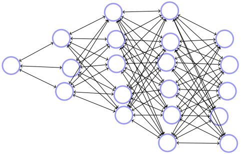
numerous other pioneering works...

(beyond this course)

Pomerleau, Knowledge-based training of artificial neural networks for autonomous robot driving, 1993

What are ANNs?

- An ANN is a network of basic computational units called (artificial) neurons



- The connection between two neurons may allow information to be sent in one direction or in both directions.
- Each neuron receives inputs along the incoming connections, and performs computes an output using simple transformations.
  - Typically, output is a simple transformation of a linear function of the inputs.
- We cover basic ANNs in this course.
  - More about ANNs: STAT3007 Deep Learning

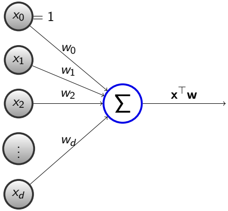
Simplest Neural Networks

Linear regression

- A linear regression model computes the function

  $$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x},$$

  where the input $\mathbf{x} = (1, x_1, \ldots, x_d)$, with x_i's being the features.
  - The dummy feature 1 means a bias term is included.
- This can be represented by the neural network below

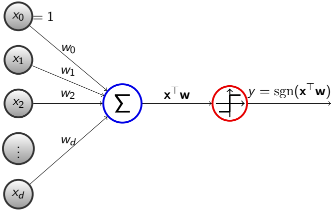


Perceptron

- The perceptron computes the function

$$f(\mathbf{x}) = \begin{cases} +1, & \mathbf{w}^\top \mathbf{x} > 0, \\ -1, & \text{otherwise.} \end{cases} .$$

- This can be represented by the neural network below


$x_0 = 1$
x_1
x_2
$\vdots$
x_d
w_0
w_1
w_2
w_d
Σ
$\mathbf{x}^\top \mathbf{w}$
$y = \text{sgn}(\mathbf{x}^\top \mathbf{w})$

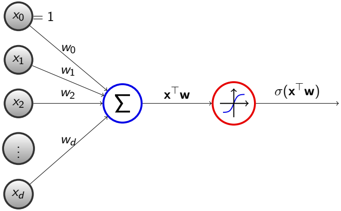
- While summation (computing $\mathbf{x}^\top \mathbf{w}$) and thresholding (of $\mathbf{x}^\top \mathbf{w}$) are shown separately in the above diagram, they are operations within a single neuron.

Logistic regression

- The (binary) logistic regression model computes

$$f(\mathbf{x}) = p(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}},$$

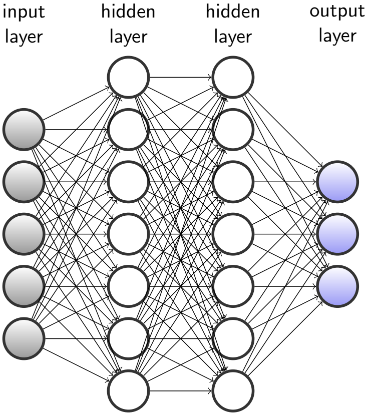
  that is, the probability that $\mathbf{x}$ is positive.
    - $\sigma(u) = \frac{1}{1+e^{-u}}$ is the logistic function.
- This can be represented by the neural network below


$x_0 = 1$
x_1
x_2
$\vdots$
x_d
w_0
w_1
w_2
w_d
Σ
$\mathbf{x}^\top \mathbf{w}$
$\sigma(\mathbf{x}^\top \mathbf{w})$

- Again, summation and the sigmoid transformation (of $\mathbf{x}^\top \mathbf{w}$) are operations within a single neuron.

- The above examples are the simplest neural networks, and the simplest multilayer perceptrons.
- They only perform one-level of computation with the raw inputs, and are very limited in their ability to express complex functional relationships.
- Multilayer perceptrons allow multi-level transformations of the inputs and can represent very complex functional relationships.

Multilayer Perceptron (MLP)

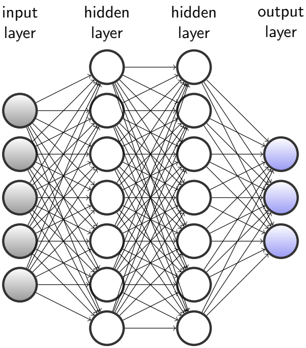


Structure of an MLP

- An MLP is also known as a multilayer feedforward neural network
  - in a feedforward neural network, the connections do not form cycles (note that each connection points from the input neuron to the output neuron).
  - in a multilayer network, the neurons are grouped into different layers
  - the dummy neuron for the bias term is often omitted
- The depth or the number of layers is the number of all layers with tunable parameters (i.e. all layers except the input layer).
- An MLP can be seen as a series of complex transformations.

Naming the layers and neurons

- The input layer is also called the first/bottom layer, and neurons in it are called input neurons/units.
- The output layer is also called the last/top layer, and neurons in it are called output neurons/units.
- Layers between the input and the output layers are called hidden layers, and neurons in them are called hidden neurons/units.
- A neural with more than one hidden layer is called a deep neural network.



- This is a 3 layer MLP, or a 2 hidden layer MLP.
- There are 5 input units, 7 hidden units for each of the two hidden layers, and 3 output units.

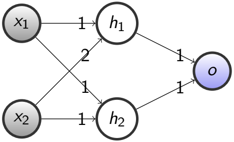
Computation in neural nets

- Each neuron first computes the weighted input sum (including the bias term), then applies an activation/transfer function to transform the weighted input sum to an output.
- Some activation functions
  - Threshold activation $I(u) = \begin{cases} 1, & u > 0, \\ -1, & \text{otherwise.} \end{cases}$.
  - Sigmoid activation $\sigma(u) = \frac{1}{1+e^{-u}}$.
  - The rectifier $(u)_+ = \max(0, u)$. A linear unit using the rectifier activation is called a ReLU (rectified linear unit).

- For a vector $\mathbf{u} = (u_1, \ldots, u_d)$, and an activation function a, we shall use $a(\mathbf{u})$ to denote $(a(u_1), \ldots, a(u_d))$, that is, we apply the activation function to each component of $\mathbf{u}$.

Example

- Consider the following MLP, with sigmoid hidden units and identity output activation, and weights shown on the edges.



- Then the output o is obtained using the following computation

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \sigma\left(W_1 \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right), \qquad o = W_2 \begin{pmatrix} h_1 \\ h_2 \end{pmatrix},$$

where $W_1 = \begin{pmatrix} w_{1,11} & w_{1,12} \\ w_{1,21} & w_{1,22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}$, and
$W_2 = \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$.

- The function computed by the network can be written as

$$o = W_2\sigma\left(W_1\begin{pmatrix}x_1\\x_2\end{pmatrix}\right) = \frac{1}{1+e^{-x_1-2x_2}} + \frac{1}{1+e^{-x_1-x_2}}.$$

- When $x_1 = 1$, $x_2 = 1$, we have

$$\begin{pmatrix}h_1\\h_2\end{pmatrix} = \begin{pmatrix}\sigma(3)\\\sigma(2)\end{pmatrix}, \quad o = h_1 + h_2 = \sigma(3) + \sigma(2) \approx 1.83$$

Regression Networks

- Consider a regression problem of predicting the value of an input $\mathbf{x} \in \mathbf{R}^d$.
- We usually design a neural net with a single output $o = f_{\mathbf{w}}(\mathbf{x})$, where $\mathbf{w}$ consists of all the network parameters.
- Given a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathbf{R}^d \times \mathbf{R}$, training/learning the neural net often amounts to minimizing the quadratic loss, or mean squared error (MSE)

$$\min_{\mathbf{w}} L(\mathbf{w}) \overset{def}{=} \frac{1}{n} \sum_{i=1}^{n} (f_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2.$$

Classification Networks

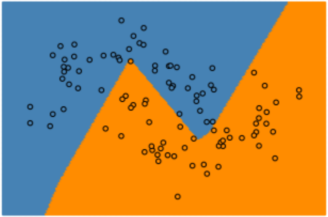
- Consider a classification problem of classifying an input $\mathbf{x} \in \mathbf{R}^d$ to one of C classes.
- We usually design a neural net $f_{\mathbf{w}} : \mathbf{R}^d \to \mathbf{R}^C$, with each output being the score for a class.
- The class with the largest score is the predicted class.
- Given a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathbf{R}^d \times [C]$, learning the neural net often amounts to minimizing the log-loss

$$\min_{\mathbf{w}} L(\mathbf{w}) \overset{def}{=} \frac{1}{n} \sum_{i=1}^{n} - \ln p(y_i \mid \mathbf{x}_i, \mathbf{w}),$$

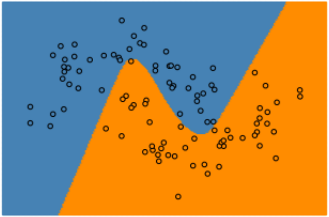
where $p(y_i \mid \mathbf{x}_i, \mathbf{w}) = e^{o_{i,y_i}} / \sum_{c=1}^{C} e^{o_{ic}}$, with $(o_{i1}, \ldots, o_{iC})$ being the network's output vector for $\mathbf{x}_i$.

Revisiting Two Moons

- Decision boundary for a single hidden layer MLP with 10 hidden units and 2 linear output units



ReLU hidden units



tanh hidden units

- When using ReLU hidden units, the decision boundary is piecewise linear and have sharp corners.
- When using tanh hidden units, the decision boundary is much smoother.

Checking Your Understanding

Which of the following statement is correct? (Multiple choice)

(a) In a multilayer perceptron, neurons are organized into several layers with connections between adjacent layers only.

(b) A classification network usually has only one output neuron.

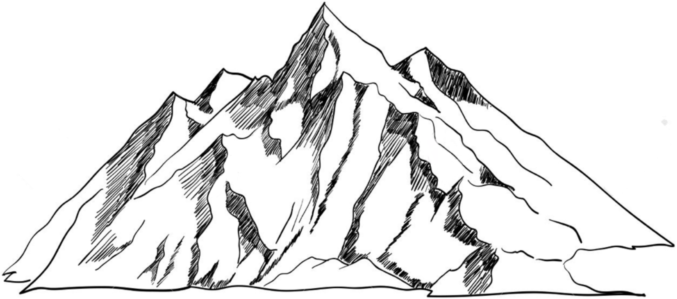
(c) A neuron in a neural network always outputs the weighted input sum.

(d) The linear regression model can be viewed as a neural net.

Neural Network Learning

- To solve the optimization problem of minimizing the loss $L(\mathbf{w})$ for learning the neural net, we typically use numerical methods.
- In particular, we often use gradient-based methods to minimize the loss function.

The Hill Climber Analogy



- You want to get down a hill, but you have very limited visibility due to a heavy fog.
- You have a tool that allows you to measure the steepness of the hill at your location along any direction.
- Using the tool takes a lot of time, and you do not want to use it too frequently.

- We can choose to move down along the steepest downhill direction.
- However, we need to determine how much we get down along that direction each time after we measure steepness so that we do not go off the track and then possibly go downhill instead.

Gradient Descent

- Gradient descent is the most basic gradient-based method for minimizing a function $L(\mathbf{w})$.
- Assume that we start from some $\mathbf{w}_0$, then at iteration $t \geq 0$, we compute the next iterate $\mathbf{w}_{t+1}$ using

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla L(\mathbf{w}_t),$$

  where $\eta_t \geq 0$ is a step size to be chosen.
- The step size is often called the learning rate when $L(\mathbf{w})$ is an objective function in machine learning.

Why does gradient descent work

- Using the first order Taylor series expansion, for any small vector d,

$$L(\mathbf{w} + d) \approx L(\mathbf{w}) + d^\top \nabla L(\mathbf{w}).$$

- Hence for small η, we have

$$\begin{aligned} L(\mathbf{w}_{t+1}) &= L(\mathbf{w}_t - \eta_t \nabla L(\mathbf{w}_t)) \\ &\approx L(\mathbf{w}_t) + (-\eta_t \nabla L(\mathbf{w}_t))^\top \nabla L(\mathbf{w}_t) \\ &= L(\mathbf{w}_t) - \eta_t \|\nabla L(\mathbf{w}_t)\|_2^2 \\ &< L(\mathbf{w}_t), \end{aligned}$$

if $\nabla L(\mathbf{w}) \neq 0$. That is, the function value decreases if we move along the negative gradient direction by a small step.

Step size matters!

- Consider the following minimization problem

$$\min_{x \in \mathbf{R}} f(x, y) = x^2 + y^2$$

- Clearly, the minimizer is $(0, 0)$.
- We consider how gradient descent works starting from $(2, 3)$ using different step sizes.

- Constant step size $\eta_s = 0.25$ for all s.
  - $(x_1, y_1) = (2, 3)$.
  - $(x_2, y_2) = (x_1, y_1) - 0.25 \cdot 2(x_1, y_1) = 0.5(x_1, y_1)$.
  - $(x_3, y_3) = (x_2, y_2) - 0.25 \cdot 2(x_2, y_2) = 0.5(x_2, y_2) = 0.5^2(x_1, y_1)$.
  - $\dots$
  - $(x_s, y_s) = 0.5^{s-1}(x_1, y_1) \to (0, 0)$ as $s \to \infty$.
- In this case, we never find the minimizer in finitely many iterations, but we can find a solution that is arbitrarily close to the minimizer after sufficiently many iterations.

- Constant step size $\eta_s = 1$ for all s.
  - $(x_1, y_1) = (2, 3)$.
  - $(x_2, y_2) = (x_1, y_1) - 1 \cdot 2(x_1, y_1) = -(x_1, y_1)$.
  - $(x_3, y_3) = (x_2, y_2) - 1 \cdot 2(x_2, y_2) = -(x_2, y_2) = (x_1, y_1)$.
  - $\ldots$
  - $(x_s, y_s) = (-1)^{s-1}(x_1, y_1)$ does not converge to the minimizer.
- We never encounter a solution that is close to the minimizer in this case!

- Step size or learning rate determines whether gradient descent converges or not, and how fast it converges. To make gradient descent converge fast,
  - The step size need to be small enough to guarantee function value is decreasing.
  - The step size need to be large enough so that there is a sufficient decrease.

  It is often not easy to choose a good step size.
- A simple guideline: try numbers like 1, 0.1, 0.01,..., ignore those leading to divergence, and choose the one with best generalization performance among the remaining ones.

Stochastic Gradient Descent (SGD)

- In machine learning, the loss $L(\mathbf{w})$ often has the form

$$L(\mathbf{w}) = \frac{1}{n}\sum_i L_i(\mathbf{w}),$$

  where $L_i(\mathbf{w})$ measures how the model $\mathbf{w}$ fits example i.
  - e.g., in regression , $L_i(\mathbf{w}) = (f_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$.
- In SGD, instead of using full gradients computed on the whole dataset, we use stochastic gradients computed on random selected examples.

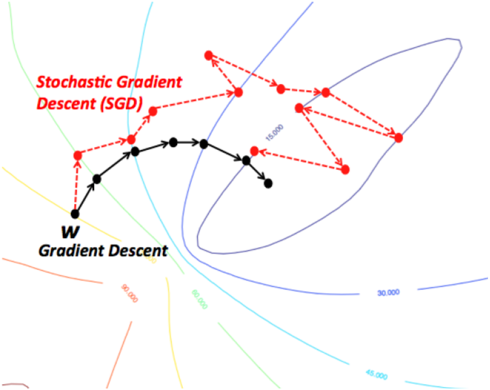
- That is, we update $\mathbf{w}_t$ to

  $$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \tilde{g}_t,$$

  where $\tilde{g}_t = \nabla L_i(\mathbf{w})$ with i randomly sampled from $1, \ldots, n$.
- We often use the mini-batch version of SGD, in which we set

  $$\tilde{g}_t = \frac{1}{|S|} \sum_{i \in S} \nabla L_i(\mathbf{w}),$$

  where S is a random subset of $1, \ldots, n$.
- While SGD can jump around in the solution space, the mini-batch version is much more stable.



- Gradient descent has a smooth trajectory, while SGD has a zigzagging trajectory.

- SGD is often computationally attractive when the dataset is very large.
- In practice, if the dataset has a lot of redundancy, SGD is able to find a good solution quickly even though it uses only a subset of the examples (typically the case for large datasets).

More on Gradient-based Learning

- Gradient descent is the simplest algorithm to use gradients for optimization.
- There are many more sophisticated algorithms
    - momentum, AdaDelta, Adam... (beyond this course)

Checking Your Understanding

Which of the following statement is correct? (Multiple choice)

(a) Neural network learning is often formulated as an optimization problem.

(b) Gradient descent minimizes a function by moving along the steepest descent direction.

(c) The best step size for gradient descent is 1.

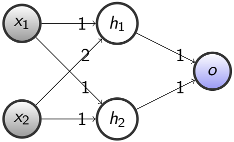
(d) Several early neural net training algorithms are special cases of SGD.

Gradient Computation for MLPs

- To apply gradient descent or SGD to minimize the loss function for a neural net, we need to be able to compute the gradient of the loss.
- In principle, this can be done using basic differentiation rules.

Revisiting the toy problem

- Assume that the observed output for the input $(1,1)$ is $y = 2$, and we want to minimize the squared error $L = (o - y)^2$.



- Recall: the output o is obtained using the following computation

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \sigma\left(W_1 \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right), \qquad o = W_2 \begin{pmatrix} h_1 \\ h_2 \end{pmatrix},$$

where $W_1 = \begin{pmatrix} w_{1,11} & w_{1,12} \\ w_{1,21} & w_{1,22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}$, and $W_2 = \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$.

- For $\frac{\partial L}{\partial w_{2,1}}$, using the chain rule

$$\frac{\partial L}{\partial w_{2,1}} = \frac{\partial L}{\partial o} \frac{\partial o}{\partial w_{2,1}} = 2(o - y)h_1.$$

- Derivatives like $\frac{\partial L}{\partial w_{1,11}}$ are much more complex (try it out).
- We see that even for this small MLP, it is tedious to compute the gradient of the loss function.

Backpropagation

- The backpropagation algorithm provides an efficient way to compute the gradient of the loss function of a feedforward neural net, which is essential in gradient-based learning.
- The algorithm performs a forward pass and a backward pass through the neural net
    - the forward pass propagates information from the input neurons to the output neurons to compute the outputs of all neurons
    - the backward pass propagates information from the output neurons to the input neurons to compute derivatives

- We illustrate the backpropagation algorithm on an MLP $f_{\mathbf{w}}(\mathbf{x})$
  - all hidden units are sigmoid units
  - there is one output neuron with identity activation function
  - the loss is the squared error (strictly speaking, 1/2 squared error)

$$L = \sum_i \frac{1}{2}(f_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2.$$

- Notations
  - $P(j)$: the set of parents of unit j.
  - o_i: the output of unit i. For an input neuron, o_i denotes its input.
  - w_{ij}: weight on the connection from unit i to unit j.

Forward propagation

For each neuron j,

$$o_j \leftarrow \begin{cases} \sigma(\sum_{i\in P(j)} w_{ij} o_i), & \text{if } j \text{ is not the output neuron.} \\ \sum_{i\in P(j)} w_{ij} o_i, & \text{if } j \text{ is the output neuron.} \end{cases}$$

when all input o_i's have been computed.

- we don't need to keep the neurons waiting for their inputs to be ready.
- instead, we compute the outputs one layer at a time from the input layer to the output layer (as illustrated in the small numerical example).

The backpropagation algorithm

- We need to compute the derivative g_{ij} of the loss function wrt to each weight w_{ij}
- We only need to figure out how to do this for one example $(\mathbf{x}, y)$
  - if there are multiple examples, the gradient is the sum of the individual gradients computed on these examples

```
Compute all $o_i$'s.
For the output unit $k$,
    $\delta_k \leftarrow (o_k - y).$
For each hidden unit $i$,
    $\delta_i \leftarrow o_i(1 - o_i) \sum_{j \in C(i)} w_{ij}\delta_j$
when all input $\delta_j$'s have been computed.
For each connection $(i, j)$,
    $g_{ij} \leftarrow \delta_j o_i.$
```

Derivation (optional)

- Notations
  - $L(o_k, y) = \frac{1}{2}(o_k - y)^2$ is the loss function (neuron k is output).
  - $s_j = \sum_{i \in P(j)} w_{ij} o_i$ is the weighted input sum for neuron j.
  - $\delta_i = \partial L / \partial s_i$.
- For the output unit k,

$$\delta_k = \frac{\partial L}{\partial s_k} = \frac{\partial L}{\partial o_k} \frac{\partial o_k}{\partial s_k} = (o_k - y).$$

This is because $o_k = s_k$ (identity activation).

- Using the chain rule, we have

$$\delta_i = \frac{\partial L}{\partial s_i} = \sum_{j \in C(i)} \frac{\partial L}{\partial s_j} \frac{\partial s_j}{\partial o_i} \frac{\partial o_i}{\partial s_i} = \sum_{j \in C(i)} \delta_j w_{ij} o_i (1 - o_i).$$

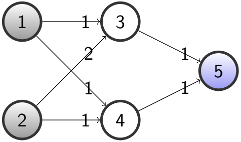
  This is because $o_i = \sigma(s_i)$ and $\sigma'(s_i) = o_i(1 - o_i)$.

- In addition, we have

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial s_j} \frac{\partial s_j}{\partial w_{ij}} = \delta_j o_i.$$

Finishing off the small MLP example

- We label the nodes as follows



- We first compute the o_i's, then δ_i's, and finally g_{ij}.

i	o_i	δ_i
1	1	-
2	1	-
3	$\sigma(3)$	$\delta_5 w_{35} o_3(1-o_3)$
4	$\sigma(2)$	$\delta_5 w_{45} o_4(1-o_4)$
5	$\sigma(3)+\sigma(2)$	$o_5 - 1$

(i,j)	g_{ij}
$(1,3)$	$o_1\delta_3$
$(2,3)$	$o_2\delta_3$
$(1,4)$	$o_1\delta_4$
$(2,4)$	$o_2\delta_4$
$(3,5)$	$o_3\delta_5$
$(4,5)$	$o_4\delta_5$

Extensions

- We can extend the backpropagation algorithm to handle different loss functions, activation functions and multiple output units.
- By choosing different loss functions and using multiple output neurons, we can train an MLP for classification and density estimation.

Automatic Differentiation (Autodiff)

- Backprop provides an efficient way to compute the gradient of the error function for a neural network.
- It is helpful to understand the algorithm but it is not easy to implement
  - you probably never want to implement it if you don't have to
  - in the old days, it is not unusual that people spent hours to derive expressions for the gradients, and then hours for implementation and debugging...

- The good news is that for practical purposes, you don't have to implement gradient computation for neural nets
  - Many machine learning software platforms now provide automatic differentiation (autodiff) tools
  - Autodiff automatically compute the gradients for you — you only need to write code to evaluate the function

Checking Your Understanding

Which of the following statement is correct? (Multiple choice)

(a) It is generally easy to manually work out the formula for the gradient of the loss of a neural net, and implement it from scratch.

(b) Backpropagation allows efficient computation of the gradient of the loss function of an MLP wrt the network parameters in a recursive manner.

(c) Autodiff allows us to implement one, get one free (implement function evaluation code, get gradient evaluation code free).

Software Frameworks

TensorFlow

Sonnet

Keras

SWIFT

theano

Caffe2

PyTorch

ConvNetJS
Deep Learning in your browser

mxnet

GLUON

DL4J

scikit learn

- Some early software frameworks have become obsolete (e.g. Theao, Caffe)
- TensorFlow, originally developed by Google Brain Team, is the most popular deep learning frameworks, with a few high-level API built on top of it (e.g. Sonnet, Keras, Swift, TFLearn)
- PyTorch, developed by Facebook, is a more recent player, but has become a main competitor of TensorFlow.
  - Simple and flexible
  - We will discuss PyTorch in this lecture.

PyTorch

- PyTorch has a very polished Python interface, and a C++ frontend.
- PyTorch provides great support for
  - Tensor computing (like NumPy), with strong GPU acceleration
  - Deep neural networks, based on autodiff.
- See `https://pytorch.org/` for details including installation instructions, tutorials, and documentation.

Neural Networks in PyTorch

- PyTorch provides several packages
  - `torch`: a general-purpose tensor package with GPU support
  - `torch.autograd`: a package for automatic differentiation
  - `torch.nn`: a neural net library with common layers and loss functions
  - `torch.optim`: contains common optimization algorithms
- We cover basics of these packages in this lecture.

Tensor Computation

```python
import torch
x, y, z = torch.zeros(3, 3), torch.ones(3, 3), torch.rand(3, 3)
print(x, y, z)
print(x + y)
print(y @ z) # matrix multiplication
print(z.int()) # convert to integer array
print(z.numpy()) # convert to numpy array
if torch.cuda.is_available(): # use GPU if available
  y, z = y.cuda(), z.cuda()
  print(y @ z)
```

Autodiff for $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$

```python
def f(x):
  return torch.dot(x, x)

x = torch.ones(2, requires_grad=True)
y = f(x)

# use the autograd library to compute all gradient information
y.backward()

# print the gradient of the function with respect to x
print(x.grad)
```

Exercise: try replacing f with your favourite function.

OLS using PyTorch

- Recall: a linear regression model computes the function

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x},$$

  where the input $\mathbf{x} = (1, x_1, \ldots, x_d)$, with x_i's being the features.
- This is a single layer MLP with one output neuron only.
- In OLS (ordinary least squares), we find $\mathbf{w}$ to minimize the MSE

$$\min_{\mathbf{w}} \frac{1}{n} \sum_i (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$$

Data

```
def regression_data(n=500, d=2):
  X = torch.rand(n, d)
  w = torch.rand(d+1)
  Y = X @ w[1:] + w[0] + torch.rand(n) * 0.1
  return X, Y

X, Y = regression_data()
```

- The output is a perturbed linear function of the inputs.

First version

```
X = torch.cat([torch.ones(X.shape[0], 1), X], dim=1) # add 1
w = torch.zeros(X.shape[1], requires_grad=True)

for i in range(200):
  loss = torch.mean((X @ w - Y)**2)
  if w.grad is not None:
      w.grad.zero_() # important: reset the stored gradient to 0
  loss.backward()
  w.data.add_(-0.5*w.grad.data)

print(w)
```

- We only use the autodiff feature in PyTorch, but control all other aspects.
- Exercise: try the above code and use the closed-form formula to compute **w**. Do you get the same answers? (You should)

Second version

```
import torch.optim as optim
from torch.autograd import Variable

X = torch.cat([torch.ones(X.shape[0], 1), X], dim=1) # add 1
w = Variable(torch.zeros(X.shape[1]), requires_grad=True)

optimizer = optim.SGD([w], lr=0.5, momentum=0)
for i in range(200):
    optimizer.zero_grad()
    loss = torch.mean((X @ w - Y)**2)
    loss.backward()
    optimizer.step()

print(w)
```

- We use the SGD optimizer provided by the optim package to zero gradient and perform gradient update.
- Variables are wrappers for tensors, and they are often interchangeble.

Third version

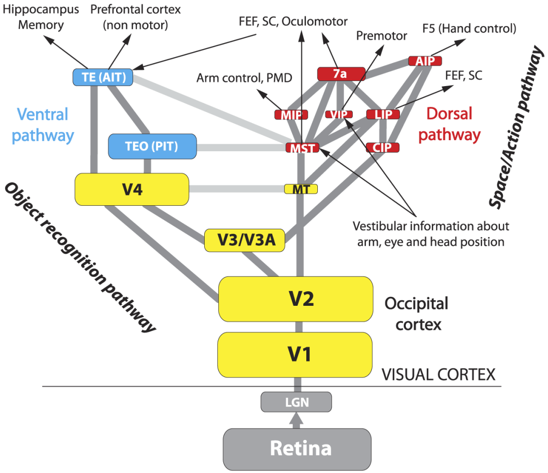
```python
import torch.optim as optim
import torch.nn as nn
from torch.nn.modules.loss import MSELoss

Y = Y.reshape(-1, 1)

net = nn.Linear(2, 1)
optimizer = optim.SGD(net.parameters(), lr=0.5, momentum=0)
mse = MSELoss()
for i in range(200):
    optimizer.zero_grad()
    loss = mse(net(X), Y)
    loss.backward()
    optimizer.step()

for param in net.parameters():
    print(param)
```

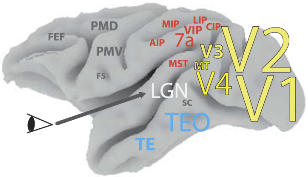
- We use the nn module to define our neural net for OLS, and use the builtin MSE loss to compute loss.

Why Deep Architectures?

- It is known that any function can be approximated arbitrarily well by a single hidden layer MLP (universal approximation theorems).
- Why do we still need to care about deep neural networks?

Inspiration from Nature





The primate visual cortex is hierarchical

Kruger et al., Deep hierarchies in the primate visual cortex: What can we learn for computer vision?, 2013

Deeper Can Be More Compact

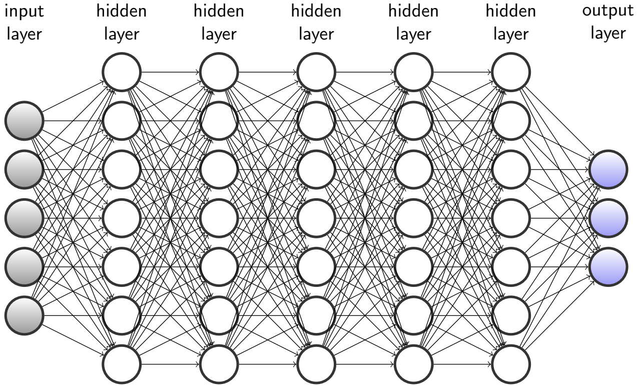
- When a function can be compactly represented by a deep network, it may need a very large shallow network to represent it.
- *E.g. There are functions computable with a depth k network consisting of a polynomially many perceptron units that require exponentially many perceptron units when using a depth $k-1$ network.*

Features: Engineering to Learning

Traditional learning: handcrafted features + classifier learning

- Many traditional learning algorithms can be seen as neural networks.
- They build classifiers using *handcrafted* features.

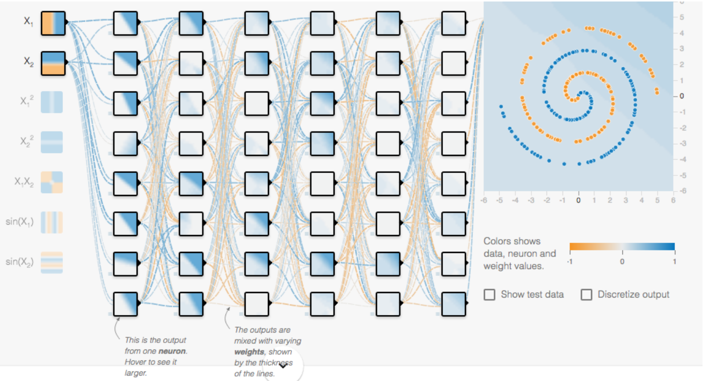
Deep learning: feature learning + classifer learning



- Deep learning uses deep architectures to additionally learn features.
- Deeper layers build abstract representations of previous layers.

  e.g. pixels → *edges* → *noses, eyes, ears* → *face*

A Demo



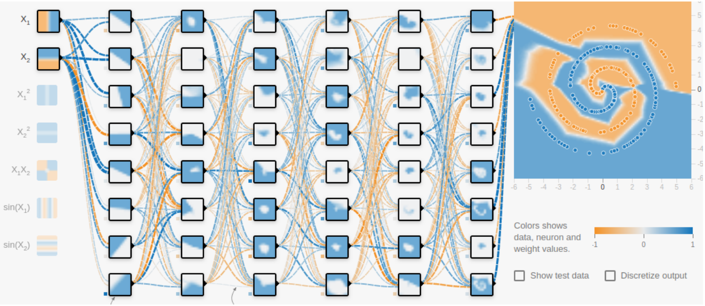
This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

- We want to distinguish points on two spirals.
- Each unit can be visualized by drawing a heat map for its output.
- Try different # of hidden layers: 1, 2, 3, 4, 5, 6.



- This trained 6-layer MLP is able to learn fairly complex decision boundaries.
- While neurons in shallow layers represent simple features (e.g. straight lines), neurons in deeper layers pick up useful high-level features (e.g. parts of the spirals).

Checking Your Understanding

Which of the following statement is correct? (Multiple choice)

(a) PyTorch supports tensor computing and deep neural nets.

(b) Deep neural nets can possibly learn complex features.

(c) Deep neural nets are always larger than an equivalent shallower networks.

What You Need to Know

- Multilayer perceptrons (aka multilayer feedforward networks)
  - Specifying an MLP: structure and activation function
  - Forward propagation (compute output for a given input)
  - Backpropagation for gradient computation
- Gradient-based learning
  - Gradient descent and stochastic gradient descent
  - Backpropagation and automatic differentiation
- Implementing neural nets using PyTorch: `torch`, `torch.nn`, `torch.autograd`, `torch.optim`
- Motivations for deep networks: inspiration from nature, more compact representation, feature learning.