INTRODUCTION & MOTIVATION
data

labels
$y$ is continuous

regression

$y$ is binary or categorical

classification
classification example

**logistic regression**

regress to the logistic transform

*linear* decision boundary

\[
\log \frac{p(y = 1|x)}{1 - p(y = 1|x)} = w^\top x + b
\]

\[
\rightarrow p(y = 1|x) = \frac{1}{1 + e^{-(w^\top x + b)}}
\]

minimize the *binary cross entropy* loss function \( L \) to find the optimal \( w \) and \( b \).

**gradient descent**

\[
w \leftarrow w - \alpha \nabla_w L
\]

\[
b \leftarrow b - \alpha \frac{\partial L}{\partial b}
\]
we need a **non-linear** decision boundary

**option 1:** use non-linear terms, expand $\mathbf{x}$ and $\mathbf{w}$

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$
classification example

\[ x = (x_1, x_2) \]

\[ y = 0 \]
\[ y = 1 \]

we need a **non-linear** decision boundary

**option 1:** use non-linear terms, expand \( x \) and \( w \)

\[ (x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1x_2, x_1, x_2) \]

**option 2:** use multiple linear decision boundaries to compose a non-linear boundary

in both cases, transform the data into a representation that is linearly separable
boolean operations

AND and OR are both linearly separable

XOR is not linearly separable, but can be separated using AND and OR
boolean operations

building XOR from AND and OR

composing non-linear boundaries from linear boundaries
recapitulation

to fit more complex data, we need more expressive **non-linear** functions

we can form non-linear functions by composing stages of processing

**depth**: the number of stages of processing

**deep learning**: learning functions with multiple stages of processing
wait...why not just use non-linear terms?

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

you certainly can!

but we will see that with enough stages of linear boundaries, we can approximate any non-linear function
artificial neuron

input features

\[ \sum \]

weights

sum

non-linearity

output feature
weights

1

x_1

x_2

\ldots

x_M

\text{sums}

\non-linearities

input features
weights

sums

non-linearities

output features

input features

1

$x_1$

$x_2$

... 

$x_M$
input features

weights

sums non-linearities

output features

\[ x_1, x_2, \ldots, x_M \]
multiple neurons form a **layer**

![Diagram of neural network showing multiple neurons forming a layer with input features, weights, sums, non-linearities, and output features.](image-url)
input features

\[ x_1 \quad x_2 \quad \ldots \quad x_M \]

weights

sums

non-linearities

1

\[
\begin{align*}
\sum & \rightarrow f \\
\sum & \rightarrow f \\
\sum & \rightarrow f
\end{align*}
\]
input features

weights

sums

non-linearities

hidden features

weights

sums

non-linearities

input features
multiple layers form a network
artificial neuron: weighted sum and non-linearity

\[ s = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^T \mathbf{x} \]

\[ h = \sigma(s) \]
artificial neuron: weighted sum and non-linearity
layer: parallelized weighted sum and non-linearity

One sum per weight vector \( s_j = w_j^\top x \longrightarrow s = W^\top x \) vector of sums from weight matrix

\[ h = \sigma(s) \]
Layer: parallelized weighted sum and non-linearity

\[
\begin{align*}
\text{input features} & \rightarrow \text{sums} & \text{non-linearities} \\
1, x_1, x_2, \ldots, x_M & \rightarrow \sum & \rightarrow \sigma
\end{align*}
\]
network: sequence of parallelized weighted sums and non-linearities

Define $x^{(0)} \equiv x$, $x^{(1)} \equiv h$, ETC.

1st layer

$s^{(1)} = W^{(1)} \tau x^{(0)}$

$x^{(1)} = \sigma(s^{(1)})$

2nd layer

$s^{(2)} = W^{(2)} \tau x^{(1)}$

$x^{(2)} = \sigma(s^{(2)})$

...
network: sequence of parallelized weighted sums and non-linearities

\[ \text{output} = \sigma \left( \ldots \sigma \left( \sigma \left( \ldots \right) \right) \right) \ldots \]
we have a method for building **expressive** non-linear functions

deep networks are *universal function approximators* (Hornik, 1991)

→ with enough units & layers, can approximate *any* function
reinterpretation

the dot product is the distance between a point and a plane

each artificial neuron defines a (hyper)plane:

\[ 0 = w_0 + w_1 x_1 + w_2 x_2 + \ldots w_M x_M \]

calculating the weighted sum corresponds to finding the shortest distance between the input point and the weight hyperplane
reinterpretation

the non-linearity transforms this distance, creating a field that changes non-linearly with distance
reinterpretation

a weight vector therefore becomes a \textit{filter} if its hyperplane faces a cluster of points within a region or subregion

the unit selects for the abstract feature shared by the cluster of points
at each stage,
1. cut the space up with hyperplanes
2. evaluate distance of each point to each hyperplane
3. transform these distances according to non-linear function
4. transformed distances become points in new space

repeat until the data are sufficiently *linearized*

→ can separate clusters with hyperplanes
neural networks are functions / function approximators

their nested (deep) structure enables a broader set of functions

output = NN(inputs)

= Layer_L(Layer_{L-1}(\ldots Layer_1(inputs)\ldots ))

can approximate a variety of functions,
particularly *conditional probability distributions*
BACKPROPAGATION
neural networks are universal function approximators, but we still must find an optimal approximating function.

We do so by adjusting the weights.
learning as *optimization*

to learn the weights, we need the **derivative** of the loss w.r.t. the weight
i.e. “how should the weight be updated to decrease the loss?”

\[ w = w - \alpha \frac{\partial \mathcal{L}}{\partial w} \]

with multiple weights, we need the **gradient** of the loss w.r.t. the weights
\[ \mathbf{w} = \mathbf{w} - \alpha \nabla \mathbf{w} \mathcal{L} \]
backpropagation

A neural network defines a function of composed operations

\[ f_L(w_L, f_{L-1}(w_{L-1}, \ldots f_1(w_1, x) \ldots )) \]

and the loss \( \mathcal{L} \) is a function of the network output.

Use chain rule to calculate gradients.

**Chain rule example**

\[ y = w_2 e^{w_1 x} \]

**Input** \( x \) \hspace{1cm} **Output** \( y \) \hspace{1cm} **Parameters** \( w_1, w_2 \)

Evaluate parameter derivatives:

\[ \frac{\partial y}{\partial w_1}, \frac{\partial y}{\partial w_2} \]

Define

\[ v \equiv e^{w_1 x} \rightarrow y = w_2 v \]
\[ u \equiv w_1 x \rightarrow v = e^u \]

Then

\[ \frac{\partial y}{\partial w_2} = v = e^{w_1 x} \]
\[ \frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial v} \frac{\partial v}{\partial u} \frac{\partial u}{\partial w_1} = w_2 \cdot e^{w_1 x} \cdot x \]
backpropagation

**recall**

1st layer

\[ s^{(1)} = W^{(1)T} x^{(0)} \]
\[ x^{(1)} = \sigma(s^{(1)}) \]

2nd layer

\[ s^{(2)} = W^{(2)T} x^{(1)} \]
\[ x^{(2)} = \sigma(s^{(2)}) \]

Loss

\[ \mathcal{L} \]

To determine the chain rule ordering, we’ll draw the dependency graph.

calculate \( \nabla_{W^{(1)}} \mathcal{L} , \nabla_{W^{(2)}} \mathcal{L} , \ldots \) let’s start with the final layer: \( \nabla_{W^{(L)}} \mathcal{L} \)
backpropagation

\[ \frac{\partial L}{\partial W^{(L)}} = \frac{\partial L}{\partial x^{(L)}} \frac{\partial x^{(L)}}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial W^{(L)}} \]

depends on the form of the loss
derivative of the non-linearity

\[
\frac{\partial}{\partial W^{(L)}} (W^{(L)\top} x^{(L-1)}) = x^{(L-1)\top}
\]

\[ \nabla_W L \equiv \frac{\partial L}{\partial W^{(L)}} \] is notational convention
backpropagation

now let’s go back one more layer...

again we’ll draw the dependency graph:

$$\frac{\partial L}{\partial W^{(L-1)}} = \frac{\partial L}{\partial x^{(L)}} \frac{\partial x^{(L)}}{\partial s^{(L)}} \frac{\partial s^{(L)}}{\partial x^{(L-1)}} \frac{\partial x^{(L-1)}}{\partial s^{(L-1)}} \frac{\partial s^{(L-1)}}{\partial W^{(L-1)}}$$
backpropagation

\[
\frac{\partial \mathcal{L}}{\partial W^{(L)}} \quad \frac{\partial \mathcal{L}}{\partial W^{(L-1)}}
\]

notice that some of the same terms appear in both gradients

specifically, we can reuse \( \frac{\partial \mathcal{L}}{\partial s^{(\ell)}} \) to calculate gradients in reverse order
backpropagation

notice that some of the same terms appear in both gradients

specifically, we can reuse $\frac{\partial L}{\partial s^{(\ell)}}$ to calculate gradients in reverse order
backpropagation

BACKPROPAGATION ALGORITHM
BACKPROPAGATION ALGORITHM

Calculate $\frac{\partial L}{\partial W^{(L)}}$
backpropagation

**BACKPROPAGATION ALGORITHM**

Calculate $\frac{\partial L}{\partial W^{(L)}}$

Store $\frac{\partial L}{\partial s^{(L)}}$
BACKPROPAGATION ALGORITHM

calculate \( \frac{\partial L}{\partial W^{(L)}} \)

store \( \frac{\partial L}{\partial s^{(L)}} \)

for \( \ell = [L - 1, \ldots, 1] \)

use \( \frac{\partial L}{\partial s^{(\ell+1)}} \) to calculate \( \frac{\partial L}{\partial W^{(\ell)}} \)
backpropagation

**BACKPROPAGATION ALGORITHM**

calculate \( \frac{\partial L}{\partial W^{(L)}} \)

store \( \frac{\partial L}{\partial s^{(L)}} \)

for \( \ell = [L - 1, \ldots, 1] \)

use \( \frac{\partial L}{\partial s^{(\ell+1)}} \) to calculate \( \frac{\partial L}{\partial W^{(\ell)}} \)

store \( \frac{\partial L}{\partial s^{(\ell)}} \)
backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial L}{\partial W^{(L)}}$

store $\frac{\partial L}{\partial s^{(L)}}$

for $\ell = [L - 1, \ldots, 1]$

use $\frac{\partial L}{\partial s^{(\ell + 1)}}$ to calculate $\frac{\partial L}{\partial W^{(\ell)}}$

store $\frac{\partial L}{\partial s^{(\ell)}}$

return $\frac{\partial L}{\partial W^{(1)}}, \ldots, \frac{\partial L}{\partial W^{(L)}}$
recapitulation

update weights using gradient of loss

backpropagation calculates the loss gradients w.r.t. internal weights

“credit assignment” via chain rule

gradient is propagated backward through the network

most deep learning software libraries automatically calculate gradients

“automatic differentiation” or “auto-diff”

can calculate gradients for any differentiable operation
IMPLEMENTATION
parallelization

neural networks can be parallelized
- matrix multiplications
- point-wise operations

using parallel computing architectures, we can efficiently implement neural network operations
import numpy as np

def nn_layer(x, W):
    s = np.dot(W.T, x)
    return np.maximum(s, 0)  # ReLU
import numpy as np

class nn_layer(object):
    def __init__(self, num_input, num_output):
        # initialize W from uniform(-0.25, 0.25)
        self.W = np.random.rand(num_input, num_output)
        self.W = 0.5 * (self.W - 0.5)

    def __call__(self, x):
        s = np.dot(self.W.T, x)
        return np.maximum(s, 0)  # ReLU
implementation

we need to manually implement backpropagation and weight updates can be difficult for arbitrary, large computation graphs

most deep learning software libraries automatically handle this for you

TF
PyTorch
mxnet
K
Caffe2

and many more

just build the computational graph and define the loss
TIPS & TRICKS
non-linearities

the non-linearities are **essential**

*without them, the network collapses to a linear function*

$$
\begin{align*}
\mathbf{I} & = \sigma( \ldots \sigma( \mathbf{I} \mathbf{W} \sigma( \mathbf{I} \mathbf{W} \ldots ) ) ) \\
\mathbf{I} & = \mathbf{I} \mathbf{W} \ldots \mathbf{I} \mathbf{W} = \mathbf{I} \mathbf{W}^L
\end{align*}
$$

different non-linearities result in different functions and optimization surfaces
non-linearities

“old school”

logistic sigmoid

hyperbolic tangent (tanh)

saturating
derivative goes to zero at $+\infty$ and $-\infty$

“new school”

rectified linear unit (ReLU)

leaky ReLU

softplus

exponential linear unit (ELU)

non-saturating

non-zero derivative at $+\infty$ and/or $-\infty$
vanishing gradients

saturating non-linearities have small derivatives almost everywhere

in backprop, the product of many small terms (i.e. $\frac{\partial x^{(\ell)}}{\partial s^{(\ell)}}$) goes to zero

$$\frac{\partial L}{\partial W^{(\ell)}} = \frac{\partial x^{(L)}}{\partial s^{(L)}} \frac{\partial x^{(L-1)}}{\partial s^{(L-1)}} \frac{\partial x^{(\ell+1)}}{\partial s^{(\ell+1)}} \frac{\partial x^{(\ell)}}{\partial \mathbf{s}^{(\ell)}} \frac{\partial \mathbf{s}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}}$$

difficult to train very deep networks with saturating non-linearities
ReLU

\[ \text{ReLU}(x) = \max(x, 0) \]

in the positive region, ReLU does not saturate, preventing gradients from vanishing in deep networks

in the negative region, ReLU saturates at zero, resulting in ‘dead units’ where the gradient is zero

but in practice, this doesn’t seem to be a significant problem
normalization

can we prevent the gradients from saturating non-linearities from becoming too small?

keep the inputs within the dynamic range of the non-linearity

we can **normalize** the activations before applying the non-linearity

\[
s \leftarrow \frac{s - \text{shift}}{\text{scale}}
\]
batch normalization

batch norm. normalizes each layer’s activations according to the statistics of the batch

\[
s^{(\ell)} \leftarrow \gamma \frac{s^{(\ell)} - \mu_B}{\sigma_B} + \beta
\]

\(\mu_B, \sigma_B\) are the batch mean and std. deviation

\(\gamma, \beta\) are additional parameters (affine transformation)

keeps internal activations in similar range, speeding up training

adds stochasticity, improves generalization
why does batch norm. work?

**original motivation:** *internal covariate shift*

changing weights during training results in changing outputs; input to the next layer changes, making it difficult to learn

batch norm. should stabilize the activations during training

*Batch Normalization, Szegedy & Ioffe, 2015*
why does batch norm. work?

but actually...

batch norm. does not seem to significantly reduce internal covariate shift

rather, it seems that batch norm. stabilizes and smooths the optimization surface

How Does Batch Normalization Help Optimization?, Santurkar et al., 2018
regularization

**neural networks are amazingly flexible…**
given enough parameters, they can perfectly fit random noise

regularization combats overfitting
*by formalizing prior beliefs on the model or data*

---

stochasticity (uncertainty)

**batch norm**

**SGD**

**dropout**

**constraints**

**early stopping**

**weight penalties**
initialization

learning is formulated as an **optimization** problem, which can be sensitive to **initial conditions**

"causes the network to blow up and/or not learn"

common strategies for weight initialization:

\[
\begin{align*}
  w & \sim U(-a, a) \\
  \text{uniform} & \\
  w & \sim \mathcal{N}(0, \sigma) \\
  \text{Gaussian}
\end{align*}
\]
optimization

stochastic gradient descent (SGD): \[ w = w - \alpha \tilde{\nabla}_w \mathcal{L} \]

use stochastic gradient estimate to descend the surface of the loss function

recent variants use additional terms to maintain “memory” of previous gradient information and scale gradients per parameter
optimization

stochastic gradient descent (SGD): \( w = w - \alpha \nabla_w \mathcal{L} \)

use *stochastic gradient* estimate to *descend* the surface of the loss function

recent variants use additional terms to maintain "memory" of previous gradient information and scale gradients per parameter

local minima and saddle points are largely not an issue in many dimensions, can move in exponentially more directions

connectivity

sequential connectivity: *information must flow through the entire sequence to reach the output*

![Sequential Connectivity Diagram]

information may not be able to propagate easily

→ *make shorter paths to output*

residual & highway connections

dense (concatenated) connections

*Deep residual learning for image recognition, He et al., 2016*

*Highway networks, Srivastava et al., 2015*

*Densely connected convolutional networks, Huang et al., 2017*
A BUFFET OF IDEAS
**soft attention**

\[ x_{\text{att.}} = a \odot x \]

re-weighting

**hard attention**

\[ x_{\text{att.}} = x[a] \]

extraction

---

**Recurrent Models of Visual Attention**, Mnih et al., 2014

**Show, Attend and Tell**, Xu et al., 2015

---

**Recurrent Models of Visual Attention**, Mnih et al., 2014
gradients of non-differentiable operations

Examples of non-differentiable operations

- **Stochastic operations**
  - $z \sim \mathcal{N}(z; \mu, \sigma^2)$
  - $z \sim \text{Cat.}(z; \mu)$

- **Discontinuous operations**
  - Zero gradient
  - Infinite gradient

- **Non-analytical operations**
  - Agent → Action → Environment → Reward
Gradients of non-differentiable operations

\[ \nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)] = \mathbb{E}_{z \sim p(z; \theta)} [f(z) \nabla_{\theta} \log p(z; \theta)] \]

Score Function Estimator (REINFORCE)  
Williams, 1992

\[ \nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)] = \mathbb{E}_{z \sim p(z; \theta)} [f(z) \nabla_{\theta} \log p(z; \theta)] \]

Pathwise Derivative Estimator (Reparameterization)  
Kingma & Welling, 2014  
Rezende et al., 2014

\[ \nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)] = \mathbb{E}_{\epsilon \sim p(\epsilon)} [\nabla_{\theta} f(z(\epsilon; \theta))] \]

e.g. \[ z \sim \mathcal{N}(z; \mu, \sigma^2) \rightarrow z = \mu + \epsilon \cdot \sigma \]
learning to optimize

optimization is a **task**

\[ \Delta \theta = f(\theta, \nabla_\theta \mathcal{L}) \]

update estimate using current estimate and curvature

\( f \) is the **optimizer**

\( \theta \) are the parameters of the **optimizee**

**learn** to perform optimization

adapted from Andrychowicz et al., 2016
adversarial examples

current neural networks are susceptible to adversarial data examples: optimize the data away from correct output

data doesn’t change qualitatively, yet is classified incorrectly

“panda”
57.7% confidence

“gibbon”
99.3% confidence

OpenAI
Szegedy et al., 2014, Intriguing properties of neural networks

Evtimov et al.
Bayesian neural networks

maintain uncertainty in the network activations and/or weights

place prior probabilities on these quantities

prevent overfitting in low-density data regions
generalization

neural networks are incredibly flexible and can fit random noise

conventional wisdom of an abstract hierarchy of features may not hold

however, different learning behavior between fitting noise and data

Understanding deep learning requires re-thinking generalization, Zhang et al., 2017
information bottleneck

**information bottleneck theory**: maximize *mutual information* between the input and output while discarding all other input information

deep networks learn representations that compress the input while preserving the relevant information for predicting the output

---

**Schwartz-Ziv & Tishby (2017)**
quantamagazine.org
PERSPECTIVE
“Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child’s? If this were then subjected to an appropriate course of education one would obtain the adult brain.”

-Alan Turing, 1950
history

1943 - McCulloch & Pitts introduce the Threshold Logic Unit to mimic a biological neuron

1957 - Frank Rosenblatt introduces the Perceptron, with more flexible weights and a learning algorithm.

1969 - Minsky & Papert show that the perceptron is unable to learn the XOR function, essentially stopping all research on neural networks in the first “AI winter.”
1982 - Hopfield introduces Hopfield networks, a type of recurrent network that is able to store auto-associative memory states.

1985 - Sejnowski & Hinton provide a method of training restricted Boltzmann machines (RBMs), a type of unsupervised generative model.

1986 - Rumelhart, Hinton, and Williams introduce the backpropagation learning algorithm, which, in fact, had already been derived as early as 1960. Interest in neural networks increases as it is shown that non-linear functions can be learned.

1995 - Hochreiter & Schmidhuber introduce long short-term memory (LSTM), which uses gating mechanisms to read and write from a memory cell.

1995 - Hinton et al. introduce the Helmholtz machine, a generative model that uses a separate “inference model” to perform posterior inference, similar to modern auto-encoders.
history

mid-1990s - mid-2000s - Interest in neural networks fade, due to data and computational constraints as well as training difficulties (e.g. vanishing gradients). The field moves toward SVMs, kernel methods, etc. This is the second “AI winter.”

2006 - Hinton et al. introduce a method for training deep belief networks through greedy layer-wise training. This work helps to ignite the move back to neural networks, which are rebranded as “deep learning.”

mid-2000s - 2011 - Deep learning slowly begins to gain traction as methods, primarily for unsupervised pre-training of networks, are developed. Other techniques, such as non-saturating non-linearities, are introduced as well. Developments in hardware and software allow these models to be trained on GPUs, hugely speeding up the training process. However, deep learning is not yet mainstream.
2011, 2012 - Huge improvements on several machine learning benchmarks (speech recognition, computer vision) definitively show that deep learning outperforms other techniques for these tasks. The field grows enormously, dominating much of machine learning.

2012 - ? - Research in deep learning skyrockets as people join and new discoveries are made. New methods and discoveries make significant contributions to supervised learning, reinforcement learning, generative modeling, etc.

we are in a golden era for research on neural networks
deep networks are function approximators.

parameterize conditional probability distributions.

most success has been in supervised learning (fit the posterior of a latent variable using labels).

in deep (model-free) RL, can approximate an agent’s value function or policy.

deep networks are a tool, real progress depends on developing better learning algorithms.
deep learning has allowed us to extend our learning algorithms to many new and relevant domains.

"hello"

can learn more complex conditional probabilities.

deep learning has also enabled the development of new learning algorithms.

Generative Adversarial Networks (GANs)
but many (human-relevant) tasks are unsolved

progress depends on...

data  hardware  software  compute  theory
looking forward

deep learning in its current form may get replaced

\[ x \rightarrow ? \rightarrow p(y|x) \]

but only by something that allows us to more easily approximate more complex probability distributions
NEXT TIME
convolutional neural networks
&
recurrent neural networks