Overview

- **Classification**: predicting a discrete-valued target
  - Binary classification: predicting a binary-valued target
  - Multiclass classification: predicting a discrete (> 2)-valued target

- Examples of multi-class classification
  - predict the value of a handwritten digit
  - classify e-mails as spam, travel, work, personal
Multiclass Classification

Classification tasks with more than two categories:
Targets form a discrete set \( \{1, \ldots, K\} \).

It’s often more convenient to represent them as one-hot vectors, or a one-of-K encoding:

\[
t = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^K
\]

entry \( k \) is 1
Multiclass Classification

- Now there are $D$ input dimensions and $K$ output dimensions, so we need $K \times D$ weights, which we arrange as a weight matrix $W$.
- Also, we have a $K$-dimensional vector $b$ of biases.
- Linear predictions:
  $$z_k = \sum_{j=1}^{D} w_{kj} x_j + b_k \quad \text{for} \quad k = 1, 2, \ldots, K$$
- Vectorized:
  $$z = Wx + b$$
Multiclass Classification

- Predictions are like probabilities: want $1 \geq y_k \geq 0$ and $\sum_k y_k = 1$
- A natural activation function to use is the softmax function, a multivariable generalization of the logistic function:

$$y_k = \text{softmax}(z_1, \ldots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

- The inputs $z_k$ are called the logits.
- Properties:
  - Outputs are positive and sum to 1 (so they can be interpreted as probabilities)
  - If one of the $z_k$ is much larger than the others, $\text{softmax}(z)_k \approx 1$ (behaves like argmax).
  - **Exercise:** how does the case of $K = 2$ relate to the logistic function?
- Note: sometimes $\sigma(z)$ is used to denote the softmax function; in this class, it will denote the logistic function applied elementwise.
If a model outputs a vector of class probabilities, we can use cross-entropy as the loss function:

$$\mathcal{L}_{CE}(y, t) = -\sum_{k=1}^{K} t_k \log y_k$$

$$= -t^T (\log y),$$

where the log is applied elementwise.

Just like with logistic regression, we typically combine the softmax and cross-entropy into a softmax-cross-entropy function.
Multiclass Classification

- Softmax regression:
  \[ z = Wx + b \]
  \[ y = \text{softmax}(z) \]
  \[ \mathcal{L}_{\text{CE}} = -t^\top (\log y) \]

- Gradient descent updates can be derived for each row of \( W \):
  \[ \frac{\partial \mathcal{L}_{\text{CE}}}{\partial w_k} = \frac{\partial \mathcal{L}_{\text{CE}}}{\partial z_k} \cdot \frac{\partial z_k}{w_k} = (y_k - t_k) \cdot x \]
  \[ w_k \leftarrow w_k - \alpha \frac{1}{N} \sum_{i=1}^{N} (y_k^{(i)} - t_k^{(i)}) x^{(i)} \]

- Similar to linear/logistic reg (no coincidence) (verify the update)
Visually, it’s obvious that XOR is not linearly separable. But how to show this?
Limits of Linear Classification

Showing that XOR is not linearly separable (proof by contradiction)

- If two points lie in a half-space, line segment connecting them also lie in the same halfspace.

- Suppose there were some feasible weights (hypothesis). If the positive examples are in the positive half-space, then the green line segment must be as well.

- Similarly, the red line segment must lie within the negative half-space.

- But the intersection can’t lie in both half-spaces. Contradiction!
Limits of Linear Classification

- Sometimes we can overcome this limitation using feature maps, just like for linear regression. E.g., for **XOR**:

\[
\psi(x) = \begin{pmatrix}
    x_1 \\
    x_2 \\
    x_1x_2
\end{pmatrix}
\]

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$\psi_1(x)$</th>
<th>$\psi_2(x)$</th>
<th>$\psi_3(x)$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- This is linearly separable. (Try it!)

- Not a general solution: it can be hard to pick good basis functions. Instead, we’ll use neural nets to learn nonlinear hypotheses directly.
Neural Networks

...
Neurons receive input signals and accumulate voltage. After some threshold they will fire spiking responses. [Pic credit: www.moleculardevices.com]
Inspiration: The Brain

- For neural nets, we use a much simpler model neuron, or unit:

\[ y = \sigma (w^\top x + b) \]

- Compare with logistic regression: \[ y = \sigma (w^\top x + b) \]

- By throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!
We can connect lots of units together into a directed acyclic graph.

Typically, units are grouped together into layers.

This gives a feed-forward neural network. That’s in contrast to recurrent neural networks, which can have cycles.
Multilayer Perceptrons

- Each hidden layer $i$ connects $N_{i-1}$ input units to $N_i$ output units.
- In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**. We’ll consider other layer types later.
- Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.
- If we need to compute $M$ outputs from $N$ inputs, we can do so in parallel using matrix multiplication. This means we’ll be using a $M \times N$ matrix.
- The output units are a function of the input units:
  \[ y = f(x) = \phi(Wx + b) \]
  
  A multilayer network consisting of fully connected layers is called a **multilayer perceptron**. Despite the name, it has nothing to do with perceptrons!
Multilayer Perceptrons

Some activation functions:

**Identity**

\[ y = z \]

**Rectified Linear Unit (ReLU)**

\[ y = \max(0, z) \]

**Soft ReLU**

\[ y = \log(1 + e^z) \]
Multilayer Perceptrons

Some activation functions:

**Hard Threshold**

\[ y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \]

**Logistic**

\[ y = \frac{1}{1 + e^{-z}} \]

**Hyperbolic Tangent (tanh)**

\[ y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
Multilayer Perceptrons

- Each layer computes a function, so the network computes a composition of functions:

\[ h^{(1)} = f^{(1)}(x) = \phi(W^{(1)}x + b^{(1)}) \]
\[ h^{(2)} = f^{(2)}(h^{(1)}) = \phi(W^{(2)}h^{(1)} + b^{(2)}) \]
\[ \vdots \]
\[ y = f^{(L)}(h^{(L-1)}) \]

- Or more compactly:

\[ y = f^{(L)} \circ \cdots \circ f^{(1)}(x). \]

- Neural nets provide modularity: we can implement each layer’s computations as a black box.
Feature Learning

Last layer:

- If task is regression: choose
  \[ y = f^{(L)}(h^{(L-1)}) = (w^{(L)})^T h^{(L-1)} + b^{(L)} \]
- If task is binary classification: choose
  \[ y = f^{(L)}(h^{(L-1)}) = \sigma((w^{(L)})^T h^{(L-1)} + b^{(L)}) \]
- Neural nets can be viewed as a way of learning features:

![Diagram of neural network](image)

- The goal:
Feature Learning

- Suppose we’re trying to classify images of handwritten digits. Each image is represented as a vector of $28 \times 28 = 784$ pixel values.
- Each first-layer hidden unit computes $\phi(w_i^T x)$. It acts as a feature detector.
- We can visualize $w$ by reshaping it into an image. Here’s an example that responds to a diagonal stroke.
Feature Learning

Here are some of the features learned by the first hidden layer of a handwritten digit classifier:
Expressive Power

- We’ve seen that there are some functions that linear classifiers can’t represent. Are deep networks any better?

- Suppose a layer’s activation function was the identity, so the layer just computes a affine transformation of the input
  - We call this a linear layer

- Any sequence of linear layers can be equivalently represented with a single linear layer.

\[ y = \underbrace{W^{(3)}W^{(2)}W^{(1)}}_{\triangleq W'} x \]

- Deep linear networks are no more expressive than linear regression.
Expressive Power

- Multilayer feed-forward neural nets with *nonlinear* activation functions are **universal function approximators**: they can approximate any function arbitrarily well.

- This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is “almost” linear, it’s nonlinear enough.
Multilayer Perceptrons

Designing a network to classify XOR:

Assume hard threshold activation function
Multilayer Perceptrons

- $h_1$ computes $\mathbb{I}[x_1 + x_2 - 0.5 > 0]$
  - i.e. $x_1$ OR $x_2$
- $h_2$ computes $\mathbb{I}[x_1 + x_2 - 1.5 > 0]$
  - i.e. $x_1$ AND $x_2$
- $y$ computes $\mathbb{I}[h_1 - h_2 - 0.5 > 0] \equiv \mathbb{I}[h_1 + (1 - h_2) - 1.5 > 0]$
  - i.e. $h_1$ AND $(\text{NOT } h_2) = x_1$ XOR $x_2$
Expressive Power

Universality for binary inputs and targets:

- Hard threshold hidden units, linear output
- Strategy: $2^D$ hidden units, each of which responds to one particular input configuration

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

- Only requires one hidden layer, though it needs to be extremely wide.
What about the logistic activation function?

You can approximate a hard threshold by scaling up the weights and biases:

\[ y = \sigma(x) \]

\[ y = \sigma(5x) \]

This is good: logistic units are differentiable, so we can train them with gradient descent.
Expressive Power

- Limits of universality
  - You may need to represent an exponentially large network.
  - How can you find the appropriate weights to represent a given function?
  - If you can learn any function, you’ll just overfit.
  - We desire a compact representation.
Training Neural Networks with Backpropagation
Recap: Gradient Descent

- **Recall:** gradient descent moves opposite the gradient (the direction of steepest descent)

- Weight space for a multilayer neural net: one coordinate for each weight or bias of the network, in *all* the layers

- Conceptually, not any different from what we’ve seen so far — just higher dimensional and harder to visualize!

- We want to define a loss $\mathcal{L}$ and compute the gradient of the cost $d\mathcal{J}/dw$, which is the vector of partial derivatives.
  - This is the average of $d\mathcal{L}/dw$ over all the training examples, so in this lecture we focus on computing $d\mathcal{L}/dw$. 
Univariate Chain Rule

- We’ve already been using the univariate Chain Rule.
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then

$$\frac{d}{dt} f(x(t)) = \frac{df}{dx} \frac{dx}{dt}.$$
Recall: Univariate logistic least squares model

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]

Let’s compute the loss derivatives \( \frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b} \)
Univariate Chain Rule

How you would have done it in calculus class

\[ L = \frac{1}{2} (\sigma(wx + b) - t)^2 \]

\[ \frac{\partial L}{\partial w} = \frac{\partial}{\partial w} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial w} (wx + b) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) x \]

\[ \frac{\partial L}{\partial b} = \frac{\partial}{\partial b} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial b} (wx + b) \]

\[ = (\sigma(wx + b) - t) \sigma'(wx + b) \]

What are the disadvantages of this approach?
Univariate Chain Rule

A more structured way to do it

Computing the loss:

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]

Computing the derivatives:

\[ \frac{d\mathcal{L}}{dy} = y - t \]
\[ \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{dy} \frac{dy}{dz} = \frac{d\mathcal{L}}{dy} \sigma'(z) \]
\[ \frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} \frac{dz}{dw} = \frac{d\mathcal{L}}{dz} x \]
\[ \frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz} \frac{dz}{db} = \frac{d\mathcal{L}}{dz} \]

Remember, the goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.
Univariate Chain Rule

- We can diagram out the computations using a computation graph.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.

Computing the loss:

\[
\begin{align*}
z &= wx + b \\
y &= \sigma(z) \\
L &= \frac{1}{2} (y - t)^2
\end{align*}
\]
Univariate Chain Rule

A slightly more convenient notation:

- Use $\bar{y}$ to denote the derivative $d\mathcal{L}/dy$, sometimes called the error signal.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

Computing the loss:

\[
\begin{align*}
    z &= wx + b \\
    y &= \sigma(z) \\
    \mathcal{L} &= \frac{1}{2}(y - t)^2 
\end{align*}
\]

Computing the derivatives:

\[
\begin{align*}
    \bar{y} &= y - t \\
    \bar{z} &= \bar{y} \sigma'(z) \\
    \bar{w} &= \bar{z} x \\
    \bar{b} &= \bar{z}
\end{align*}
\]
Multivariate Chain Rule

**Problem:** what if the computation graph has *fan-out* $> 1$?
This requires the **multivariate Chain Rule**!

$L_2$-Regularized regression

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]
\[ \mathcal{R} = \frac{1}{2} w^2 \]
\[ \mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R} \]

Softmax regression

\[ z_\ell = \sum_j w_{\ell j} x_j + b_\ell \]
\[ y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}} \]
\[ \mathcal{L} = -\sum_k t_k \log y_k \]
Multivariate Chain Rule

- Suppose we have a function \( f(x, y) \) and functions \( x(t) \) and \( y(t) \). (All the variables here are scalar-valued.) Then

\[
\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\]

- Example:

\[
f(x, y) = y + e^{xy}
\]

\[
x(t) = \cos t
\]

\[
y(t) = t^2
\]

- Plug in to Chain Rule:

\[
\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\]

\[
= (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t
\]
Multivariable Chain Rule

- In the context of backpropagation:

\[
\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\]

- In our notation:

\[
\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}
\]
Backpropagation

Full backpropagation algorithm:
Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$v_N$ denotes the variable we’re trying to compute derivatives of (e.g. loss).

### Forward pass

For $i = 1, \ldots, N$

Compute $v_i$ as a function of $\text{Pa}(v_i)$

### Backward pass

For $i = N - 1, \ldots, 1$

$$\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_i}{\partial v_j}$$

$\overline{v_N} = 1$
Backpropagation

Example: univariate logistic least squares regression

Forward pass:

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]
\[ \mathcal{R} = \frac{1}{2} w^2 \]
\[ \mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R} \]

Backward pass:

\[ \overline{\mathcal{L}}_{\text{reg}} = 1 \]
\[ \overline{\mathcal{R}} = \overline{\mathcal{L}}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} \]
\[ = \overline{\mathcal{L}}_{\text{reg}} \lambda \]
\[ \overline{\mathcal{L}} = \overline{\mathcal{L}}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} \]
\[ = \overline{\mathcal{L}}_{\text{reg}} \]
\[ \overline{y} = \overline{\mathcal{L}} \frac{d\mathcal{L}}{dy} \]
\[ = \overline{\mathcal{L}} (y - t) \]
\[ \overline{w} = \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{d\mathcal{R}}{dw} \]
\[ = \overline{z} x + \overline{\mathcal{R}} w \]
\[ \overline{b} = \overline{z} \frac{\partial z}{\partial b} \]
\[ = \overline{z} \]
Backpropagation

Multilayer Perceptron (multiple outputs):

Forward pass:

\[ z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \]
\[ h_i = \sigma(z_i) \]
\[ y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \]
\[ \mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2 \]

Backward pass:

\[ \bar{\mathcal{L}} = 1 \]
\[ \bar{y}_k = \bar{\mathcal{L}} (y_k - t_k) \]
\[ w_{ki}^{(2)} = \bar{y}_k h_i \]
\[ b_k^{(2)} = \bar{y}_k \]
\[ \bar{h}_i = \sum_k \bar{y}_k w_{ki}^{(2)} \]
\[ \bar{z}_i = \bar{h}_i \sigma'(z_i) \]
\[ w_{ij}^{(1)} = \bar{z}_i x_j \]
\[ b_i^{(1)} = \bar{z}_i \]
Backpropagation

In vectorized form:

Forward pass:

\[ z = W^{(1)}x + b^{(1)} \]
\[ h = \sigma(z) \]
\[ y = W^{(2)}h + b^{(2)} \]
\[ \mathcal{L} = \frac{1}{2} \| y - t \|^2 \]

Backward pass:

\[ \overline{\mathcal{L}} = 1 \]
\[ \overline{y} = \overline{\mathcal{L}} (y - t) \]
\[ \overline{W^{(2)}} = \overline{y}h^\top \]
\[ \overline{b^{(2)}} = \overline{y} \]
\[ \overline{h} = W^{(2)\top} \overline{y} \]
\[ \overline{z} = \overline{h} \circ \sigma'(z) \]
\[ \overline{W^{(1)}} = \overline{zx}^\top \]
\[ \overline{b^{(1)}} = \overline{z} \]
Computational Cost

- Computational cost of forward pass: one \textbf{add-multiply operation} per weight

\[ z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \]

- Computational cost of backward pass: two add-multiply operations per weight

\[ \overline{w}_{ki}^{(2)} = \overline{y}_k h_i \]
\[ \overline{h}_i = \sum_k \overline{y}_k \overline{w}_{ki}^{(2)} \]

- Rule of thumb: the backward pass is about as expensive as two forward passes.
- For a multilayer perceptron, this means the cost is linear in the number of layers, quadratic in the number of units per layer.
Backpropagation

- Backprop is used to train the overwhelming majority of neural nets today.
  - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.

- Despite its practical success, backprop is believed to be neurally implausible.