Introduction to Machine Learning

Multilayer Perceptrons

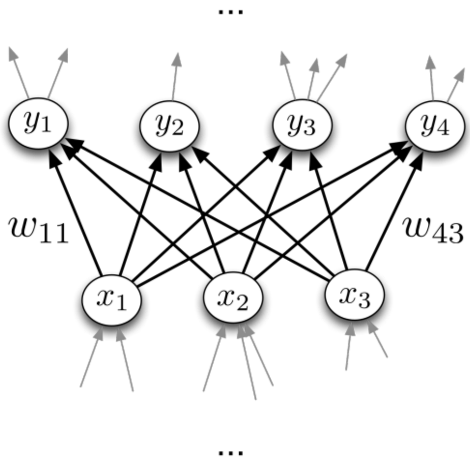
Ashley Gao

College of William & Mary

November 1, 2023

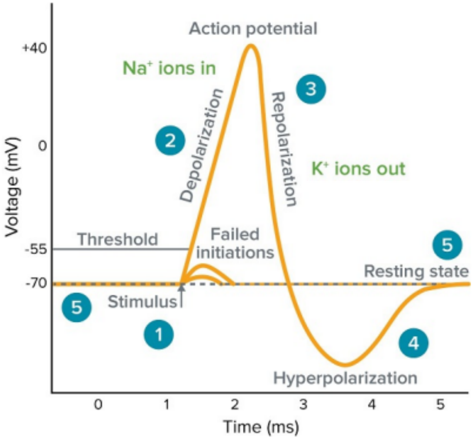
Neural Networks

- Here's an example of two layers in a deep neural network



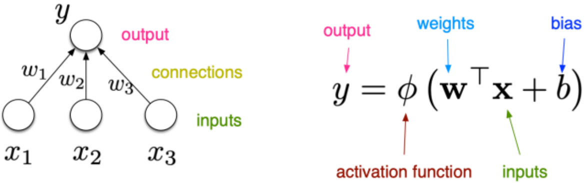
Inspiration: The Brain

- Neurons receive input signals and accumulate voltage.
- After some threshold, they will fire spiking responses.

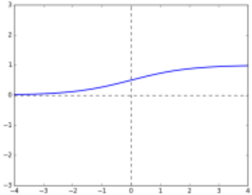


Inspiration: The Brain

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



- Compare with logistic regression: $y = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$



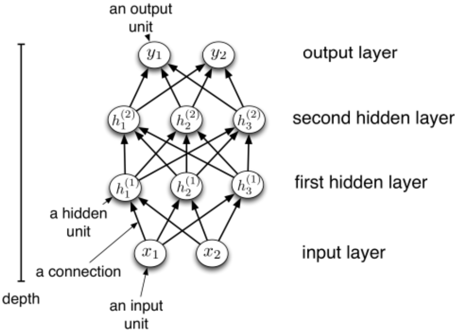
- by throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!

Overview

- Design choices so far
  - Task: regression, binary classification, multi-class classification
  - Model and Architecture: linear, SVM, decision tree ... and now neural network
  - Loss function: squared error, 0-1 loss, cross-entropy, hinge loss
  - Optimization algorithm: direct solution (calculus and linear algebra), gradient descent

Multilayer Perceptrons

- We can connect lots of units together into a directed acyclic graph
- This gives us a feed-forward neural network
    - This is in contrast to recurrent neural networks, which have cycles
    - More on RNNs later
- Typically, units are grouped together into layers



Multilayer Perceptrons

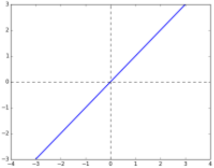
- Each layer connects N input units to M output units
- In the simplest case, all input units are connected to all output units
  - We call this a fully connected layer.
  - We will consider other layer types later
  - Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network
- We need an $M \times N$ matrix
- The output units a function of the input units

$$\boldsymbol{y} = f(\boldsymbol{x}) = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{1}$$

- A multilayer network consisting of fully connected layers is called a multilayer perceptron.

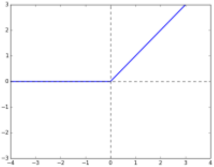
Multilayer Perceptrons

- Some activation functions:



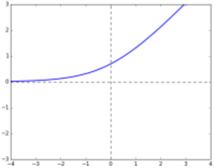
Identity

$$y = z$$



Rectified Linear Unit (ReLU)

$$y = \max(0, z)$$

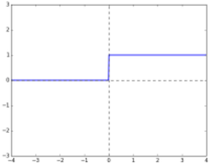


Soft ReLU

$$y = \log 1 + e^z$$

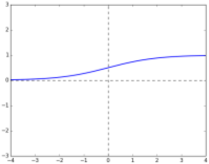
Multilayer Perceptrons

- Some activation functions:



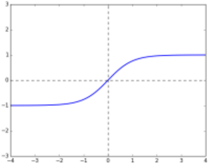
Hard Threshold

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



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

$$\boldsymbol{h}^{(1)} = f^{(1)}(\boldsymbol{x}) = \phi(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \tag{2}$$

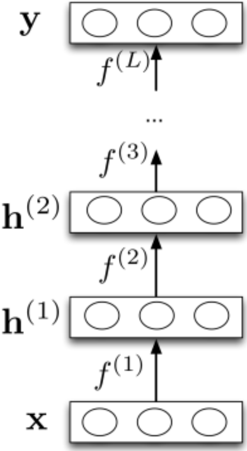
$$\boldsymbol{h}^{(2)} = f^{(2)}(\boldsymbol{h}^{(1)}) = \phi(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \tag{3}$$

$$\dots$$

$$\boldsymbol{y} = f^{(L)}(\boldsymbol{h}^{(L-1)}) \tag{4}$$

- Or more simply:

$$\boldsymbol{y} = f^{(L)} \circ \dots \circ f^{(1)}(\boldsymbol{x}) \tag{5}$$



Feature Learning

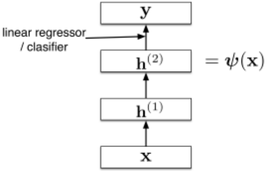
- Last layer:
  - If the task is regression: choose

$$\boldsymbol{y} = f^{(L)}(\boldsymbol{h}^{(L-1)}) = (\boldsymbol{w}^{(L)})^\top \boldsymbol{h}^{(L-1)} + b^{(L)} \tag{6}$$

  - If the task is binary classification: choose

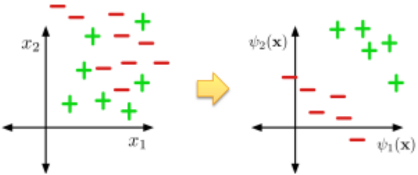
$$\boldsymbol{y} = f^{(L)}(\boldsymbol{h}^{(L-1)}) = \sigma((\boldsymbol{w}^{(L)})^\top \boldsymbol{h}^{(L-1)} + b^{(L)}) \tag{7}$$

- So neural nets can be viewed as a way of learning features



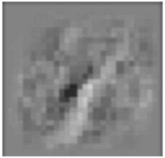
Feature Learning

- The goal of the neural nets:



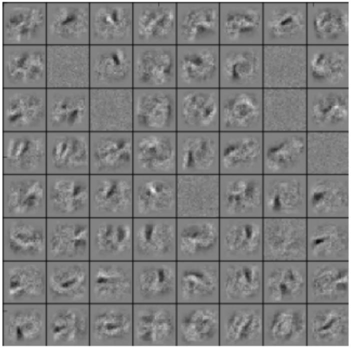
Feature Learning

- Suppose we are 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 $\phi(\boldsymbol{w}_i^\top \boldsymbol{x})$.
    - It acts as a feature detector
- We can visualize $\boldsymbol{w}$ by reshaping it into an image
    - Here is an example that corresponds to a diagonal stroke



Feature Learning

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

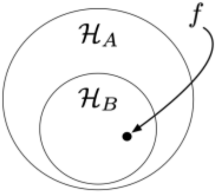


- Unlike hard-coded feature maps (e.g. in regression models), features learned by neural networks adapt to patterns in the data

Expressivity

- The idea of a hypothesis space $\mathcal{H}$ is the set of input-output mappings that can be represented by some model.
- Suppose we are deciding between two models A, B with hypothesis space $\mathcal{H}_A, \mathcal{H}_B$
- if $\mathcal{H}_B \subseteq \mathcal{H}_A$, then A is more expressive than B

A can represent any function f in $\mathcal{H}_B$.



- Some functions (XOR) can't be represented by linear classifiers. Are deep networks more expressive?

Expressivity - Linear Networks

- Suppose a layer's activation function was the identity
  - The layer just computes an 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

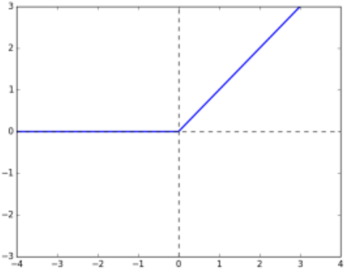
$$\boldsymbol{y} = \underbrace{\boldsymbol{W}^{(3)}\boldsymbol{W}^{(2)}\boldsymbol{W}^{(1)}}_{\triangleq \boldsymbol{W}'}\boldsymbol{x} \tag{8}$$

- Deep linear networks can only represent linear functions
- Deep linear networks are no more expressive than linear regression

Expressive Power—Non-linear Networks

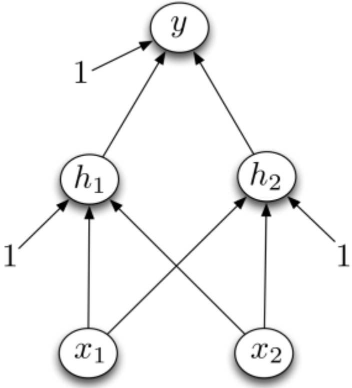
- Multilayer feed-forward neural networks with nonlinear activation functions are universal function approximators
  - They can approximate any function arbitrarily well, i.e. for any $f : \mathcal{X} \to \mathcal{T}$ there is a sequence $f_i \in \mathcal{H}$ with $f_i \to f$

  This has been shown for various activation functions (thresholds, logistic, ReLU, etc.)
  - Even though ReLU is "almost" linear, it is nonlinear enough



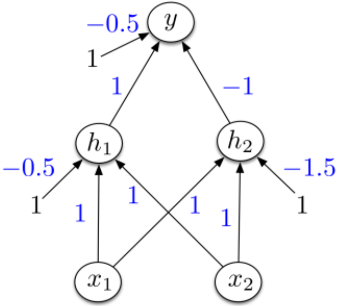
Multilayer Perceptrons

- Designing a network to classify XOR
    - Assume hard threshold activation function



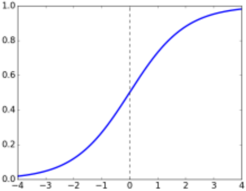
Multilayer Perceptrons

- 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 (NOT h_2) $\equiv x_1$ XOR x_2

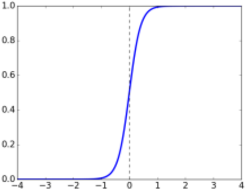


Expressivity

- What about the logistic activation function?
- You can approximate a hard threshold by scaling up the weights and biases



$$y = \sigma(x)$$

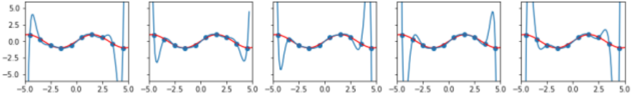


$$y = \sigma(5x)$$

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

Expressivity - What is it good for?

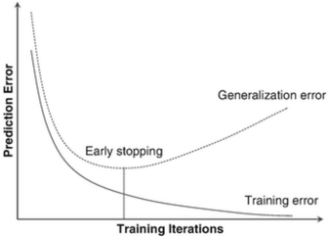
- Universality is not necessarily a golden ticket
  - You may need a very large network to represent a given function
  - How can you find the weights that represent a given function?
- Expressivity can be bad: if you can learn any function, overfitting is potentially a serious concern!
  - Recall the polynomial feature mapping. Expressivity increases with the degree M, eventually allowing multiple perfect fits to the training data



  - This motivates L^2 regularization
- Do neural networks overfit and how can we regularize them?

Regularization and Overfitting for Neural Networks

- The topic of overfitting (when & how it happens, how to regularize, etc.) for neural networks is not well-understood, even by researchers
  - In principle, you can always apply L^2 regularization
- A common approach is early stopping, or stopping training early, because overfitting typically as training progresses

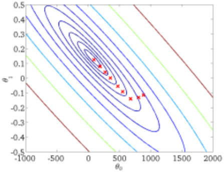


- Unlike L^2 regularization, we don't add an explicit $\mathcal{R}(\boldsymbol{\theta})$ term to our cost.

Training Neural Networks with Backpropagation

Recap: Gradient Descent

- Recall: gradient descent movies opposite the gradient (the direction of the 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 hard to visualize
- We want to define a loss $\mathcal{L}$ and compute the gradient of the cost $\frac{d\mathcal{J}}{d\mathbf{w}}$, which is the vector of partial derivatives
  - This is the average of $\frac{d\mathcal{J}}{d\mathbf{w}}$ over all the training examples, so in this lecture we focus on computing $\frac{d\mathcal{J}}{d\mathbf{w}}$

Univariate Chain Rule

- Let's now look at how we compute gradients in neural networks
- We have already been using the univariate Chain Rule
- Recall: if $f(x)$ and $x(t)$ are univariate functions, then:

$$\frac{d}{dt}f(x(t)) = \frac{df}{dx}\frac{dx}{dt} \tag{9}$$

Univariate Chain Rule

- Recall: Univariate logistic least square model

$$z = wx + b \tag{10}$$

$$y = \sigma(z) \tag{11}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{12}$$

- Let's compute the loss derivatives: $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$

Univariate Chain Rule

- Computing the loss:

$$z = wx + b \tag{13}$$

$$y = \sigma(z) \tag{14}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{15}$$

- Computing the derivatives:

$$\frac{d\mathcal{L}}{dy} = y - t \tag{16}$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{dy}\frac{dy}{dz} = \frac{d\mathcal{L}}{dy}\sigma'(z) \tag{17}$$

$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\partial\mathcal{L}}{\partial z}\frac{\partial z}{\partial w} = \frac{\partial\mathcal{L}}{\partial z}x \tag{18}$$

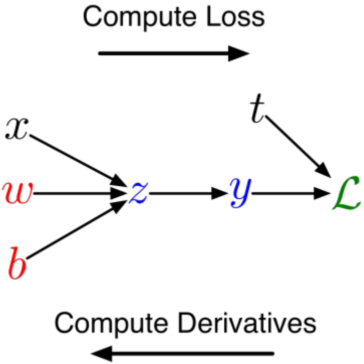
$$\frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial\mathcal{L}}{\partial z} \tag{19}$$

Recap: Gradient Descent

- We can diagram out the computation 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:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{aligned}
$$



Compute Loss

Compute Derivatives

Univariate Chain Rule

- A slightly more convenient notations:
  - Use $\bar{y}$ to denote the derivative $\frac{d\mathcal{L}}{dy}$, sometimes called the error signal
  - This emphasizes that the error signals are just values that our program is computing, rather than a mathematical operations

- Computing the loss:

$$z = wx + b \tag{20}$$

$$y = \sigma(z) \tag{21}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{22}$$

- Computing the derivatives:

$$\bar{y} = y - t \tag{23}$$

$$\bar{z} = \bar{y}\sigma'(z) \tag{24}$$

$$\bar{w} = \bar{z}x \tag{25}$$

$$\bar{b} = \bar{z} \tag{26}$$

Multivariate Chain Rule

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$.

$$\begin{tikzcd}
& x \arrow[dr] & \\
t \arrow[ur] \arrow[dr] & & f \\
& y \arrow[ur] &
\end{tikzcd}$$

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} \tag{27}$$

- Example:

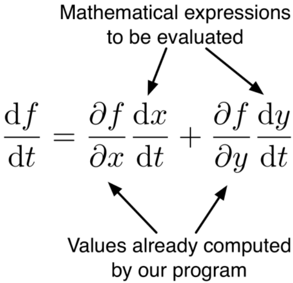
$$f(x, y) = y + e^{xy}, x(t) = \cos t, y(t) = t^2 \tag{28}$$

- Plug into the Chain Rule:

$$\frac{df}{dt} = (ye^{xy}) \cdot (-\sin t) + (1 + xe^{xy}) \cdot 2t \tag{29}$$

Multivariate Chain Rule

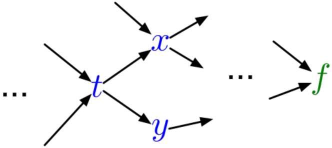
- In the context of backpropagation:



Mathematical expressions to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program

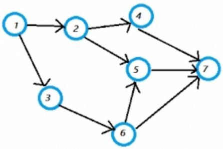


- In our notation:

$$\bar{t} = \bar{x}\frac{dx}{dt} + \bar{y}\frac{dy}{dt} \tag{30}$$

Backpropagation

- Full backpropagation algorithms:
  - 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 are trying to compute derivatives of (e.g. loss)

forward pass

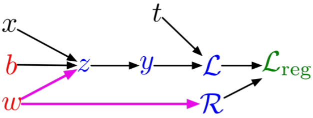
```
For $i = 1, \ldots, N$
    Compute $v_i$ as a function of $\mathrm{Pa}(v_i)$
```

backward pass

```
$\overline{v_N} = 1$
For $i = N - 1, \ldots, 1$
    $\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$
```

Backpropagation

- Example: logistic least squares regression



Forward pass:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2 \\
\mathcal{R} &= \frac{1}{2}w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda\mathcal{R}
\end{aligned}
$$

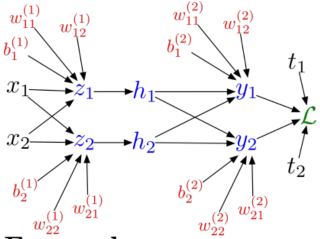
Backward pass:

$$
\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} \\
&= \overline{\mathcal{L}_{\text{reg}}}\,\lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}}\,\frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} \\
&= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}}\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \\
&= \overline{\mathcal{L}}\,(y - t)
\end{aligned}
$$

$$
\begin{aligned}
\overline{z} &= \overline{y}\,\frac{\mathrm{d}y}{\mathrm{d}z} \\
&= \overline{y}\,\sigma'(z) \\
\overline{w} &= \overline{z}\,\frac{\partial z}{\partial w} + \overline{\mathcal{R}}\frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} \\
&= \overline{z}\,x + \overline{\mathcal{R}}\,w \\
\overline{b} &= \overline{z}\,\frac{\partial z}{\partial b} \\
&= \overline{z}
\end{aligned}
$$

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:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}}\,(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\,h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

$$\overline{z_i} = \overline{h_i}\,\sigma'(z_i)$$

$$\overline{w_{ij}^{(1)}} = \overline{z_i}\,x_j$$

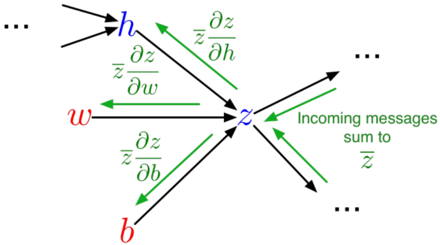
$$\overline{b_i^{(1)}} = \overline{z_i}$$

Backpropagation

- Backpropagation is the algorithm for efficiently computing gradients in neural nets
- Gradient descent with gradients computed via backpropagation is used to train the overwhelming majority of neural nets today
  - Even optimization algorithms much fancier than gradient descent use backpropagation to compute the gradients
- Despite its practical success, backpropagation is believed to be neurally impossible

Backpropagation

- Backpropagation as message passing:


...
h
$\overline{z}\frac{\partial z}{\partial h}$
$\overline{z}\frac{\partial z}{\partial w}$
...
w
z
Incoming messages
sum to
$\overline{z}$
$\overline{z}\frac{\partial z}{\partial b}$
...
b

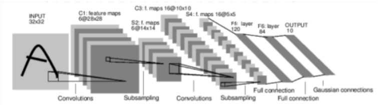
- Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents
- This provides modularity since each node has only to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph

Beyond Feed-forward Neural Networks

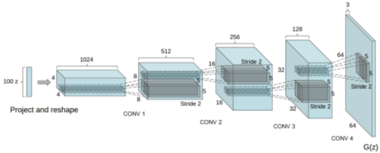
- We will talk about some of these in the next few lectures

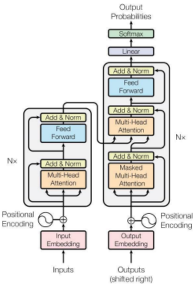
For modern applications (vision, language, games)
we use more complicated architectures.

CNN



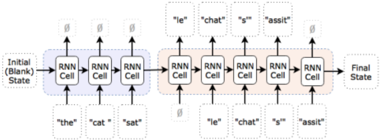
GAN

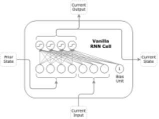




Transformer

RNN


Initial (Blank) State
RNN Cell
RNN Cell
RNN Cell
RNN Cell
RNN Cell
RNN Cell
RNN Cell
RNN Cell
Final State
"the"
"cat "
"sat"
Ø
"le"
"chat"
"s'"
"assit"
Ø
Ø
Ø
"le"
"chat"
"s'"
"assit"
Ø


Current Output
Vanilla RNN Cell
Prior State
Current State
1
Bias Unit
Current Input