Neural networks

Machine Learning (CSCI 5770G)

Faisal Z. Qureshi

http://vclab.science.ontariotechu.ca

Feed forward neural networks

- Approximate some function $y = f^*(\mathbf{x})$ by learning parameters θ s.t. $\tilde{y} = f(\mathbf{x}; \theta)$
- Feed forward neural networks can be seen as *directed acyclic graphs*

$$y = f(\mathbf{x}) = f^{(L)}(\cdots f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))))$$

- Training examples specify the output of the *last* layer
  - Network needs to figure out the inputs/outputs for the *hidden* layers

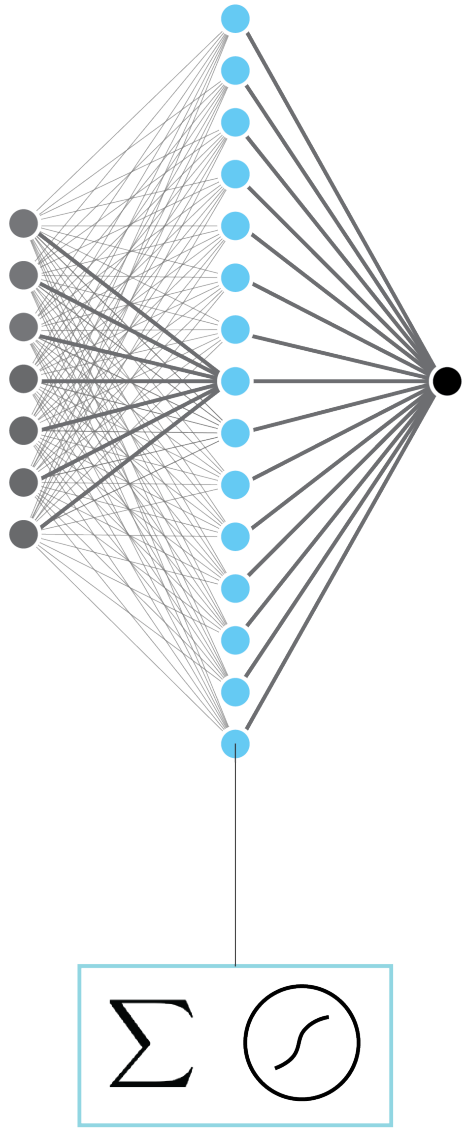
Extending linear models

How can we extend linear models?

- Specify a very general ϕ s.t. the model becomes $y = \theta^T \phi(\mathbf{x})$
  - Problem with generalization
  - Difficult to encode *prior* information needed to solve AI-level tasks
- Engineer ϕ for the task at hand
  - Tedious
  - Difficult to transfer to new tasks
- Neural networks approaches
  - $y = f(\mathbf{x}; \theta, w) = \phi(\mathbf{x}; \theta)^T w$ i.e. use parameters θ to learn ϕ and use w to map $\phi(\mathbf{x})$ to the desired output y
  - The training problem is non-convex
  - Key advantage: a designer just need to specify the right family of functions and not the exact function ϕ

Classical artificial neural networks

- Shallow and wide
- One hidden layer can represent any function
- Focus was on efficient ways to optimize (train)



Classical artificial neural networks

We can represent networks comprising a single hidden layer as

$$y = f_o\left(\mathbf{W}_{oh} f_h\left(\mathbf{W}_{ih}\mathbf{x}\right)\right).$$

Here, $\mathbf{x} \in \mathbb{R}^d$ is the d-dimensional input, $\mathbf{W}_{ih} \in \mathbb{R}^{d_h \times d}$ is the input-to-hidden-layer weight matrix, d_h is the size of the hidden layer, $\mathbf{W}_{ho} \in \mathbb{R}^{1 \times d_h}$ is the hidden-layer-to-output weight matrix, and f_h and f_o are activations functions for hidden layer and output, respectively.

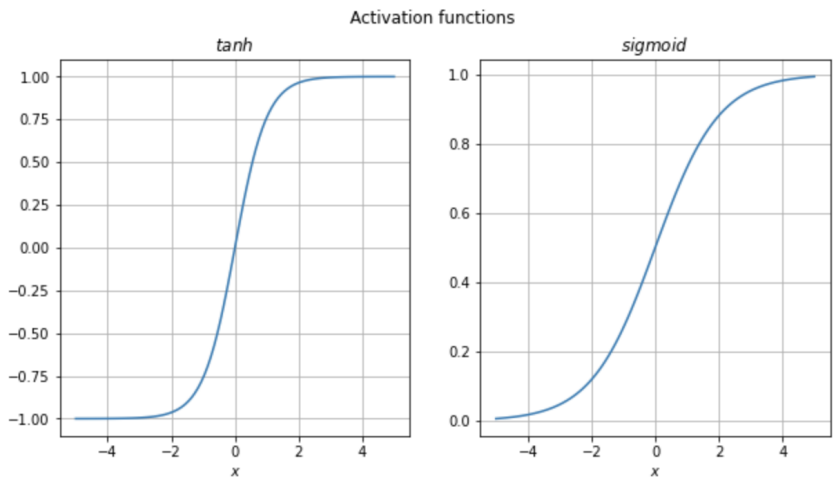
Activation functions take a vector as input and return a vector of the same size. The function is applied element-wise.

Traditionally, possible choices for f_h are:

- hyperbolic tangent; and
- sigmoid.

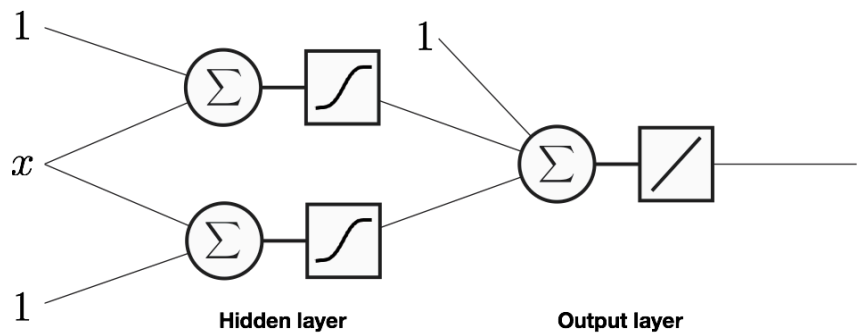
Activation functions (artificial neural networks)

- The activation functions for output:
  - Identity function for regression;
  - Sigmoid for binary classification; and
  - Softmax for multi-class classification.
- The activation functions for hidden layers:
  - tanh (allows for negative output values); and
  - sigmoid.



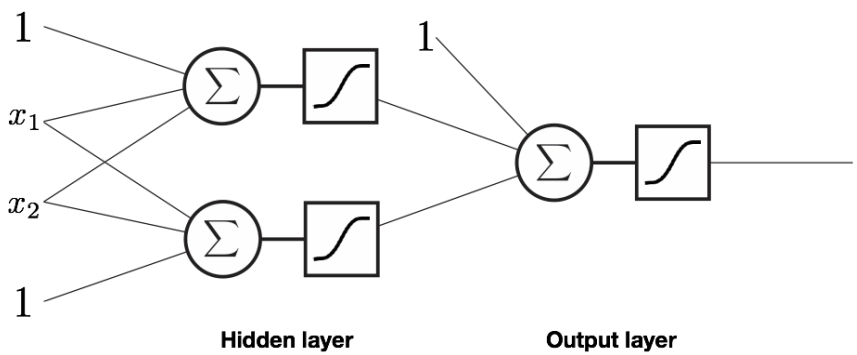
Example: a regression network

- **Input**: 1D, real numbers
- **Ouput**: 1D, real numbers
- **Hidden layer size**: 2
- **Number of weights**: 7
- **Loss**: MSE
- Probabilistic view of line fitting



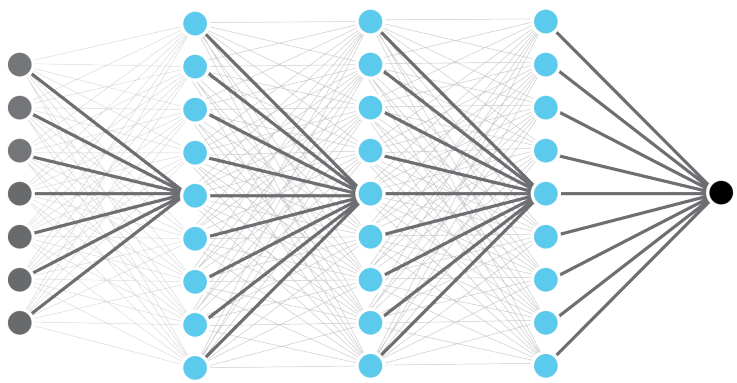
Example: a classification network

- **Input**: 2D, real numbers
- **Output**: 1D, class labels 0 or 1
- **Hidden layer size**: 2
- **Number of weights**: 9
- **Loss**: Cross-entropy
- Data likelihood under Bernoulli distribution



Current view – deep neural networks

- Multi-layer networks
  - These networks are deeper than these are wider
- Hierarchical representation
  - Reduces *semantic gap*
- Deep networks outperform humans on many tasks
- Access to data
- Advances in computer science, physics and engineering



Gradient-based learning in neural networks

- Non-linearities of neural networks render most cost functions non-convex
- Use iterative gradient based optimizers to drive cost function to lower values
- Gradient descent applied to non-convex cost functions has no guarantees is sensitive to initial conditions
  - Initialize weights to small random values
  - Initialize biases to zero or small positive values

Cost functions

- Most modern neural networks are trainined using *maximum likelihood* principle
- When parametric values defines a distribution $p(y|\mathbf{x};\theta)$ the negative log-likelihood is the cross-entropy between the training data and model predictions
- Advantage of using maximum likelihood: we get cost for free, which is $-\log p(y|\mathbf{x})$
- Gradient of the cost function must be large (and predictable)

Another advantage of using negative log likelihood as a cost function

When hidden or output units saturate, their gradients become really small, creating difficulties for gradient based learning methods. Many output units contain and $\exp()$, for example softmax, an advantage of using negative log likelihood is also that it undoes the effects of $\exp()$ preventing saturation

Output units

The role of the output units is to provide some additional transformations from the features computed by the hidden layers to complete the task at hand:

$$y = f(\mathbf{h}),$$

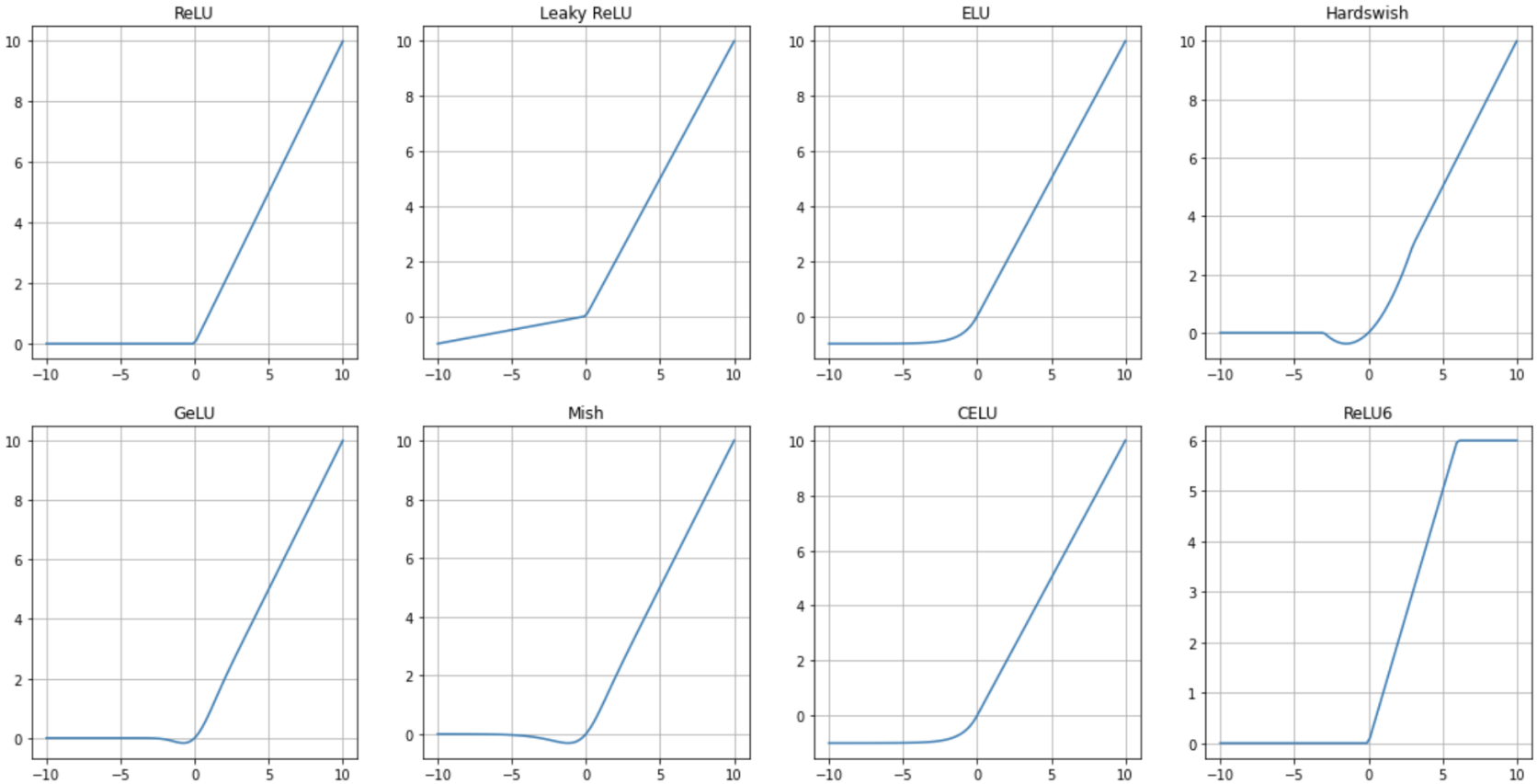
where $\mathbf{h} = f(\mathbf{x}; \theta)$ are the features computed by the hidden layer.

- Linear units
- Sigmoid units
- Softmax units

Hidden units

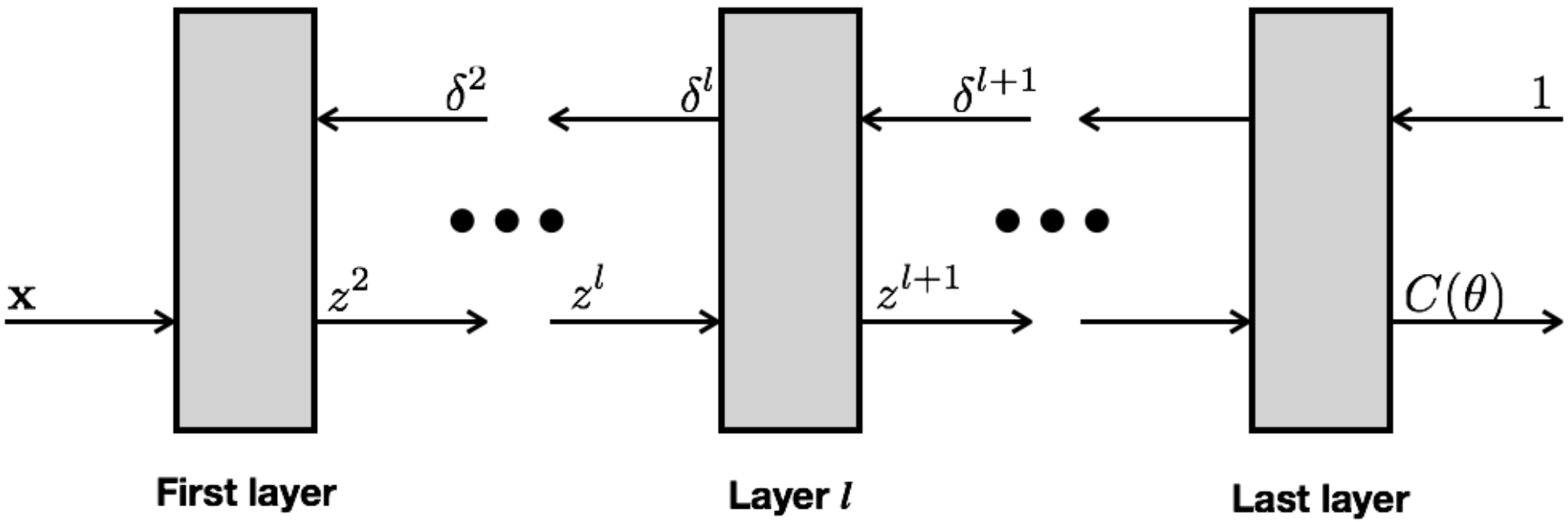
- ReLU
- Leaky ReLU
- Parametric ReLU
- Maxout
- Dropout
- Sine
- Logistic, sigmoid, hyperbolic tangent
  - Rarely used as hidden units these days, except for recurrent networks

Activation functions plots



Layered architectures

As long as we have differentiable layers, i.e., we can compute $\frac{\partial z_k^{l+1}}{\partial z_i^l}$, we can use *backpropagation* to update the parameters θ to minimize the cost $C(\theta)$.



Backpropagation

- Set z^1 equal to input $\mathbf{x}$.
- **Forward pass**:
  - Compute $z^2, z^3, \cdots$ layers $1, 2, \cdots$ activations
  - Set δ at the last layer equal to 1
- **Backward pass**:
  - Backpropagate $\delta^L, \delta^{L-1}, \cdots, \delta^2$ (all the way to first layer)
  - Compute $\nabla_\theta C(\theta)$
- Update θ
- Repeat

Layered architecture: consequences

- Compositionality
- Reuse
- Ease of constructing **your own layers**

A neural network can itself be treated as a layer within another neural network (recursion). This allows us to build new neural networks using exisitng (and sometimes pre-trained) models.

This is all good in theory, but what about practice

GPUs

- Support fast vectorized processing

Autodiff

- Techniques to evaluate the derivative of a computer program
- Autodiff example on Google Colab

Autodiff example

```
import torch
import numpy as np

def sigmoid(x):
    return 1. / (1. + torch.exp(-x))

def derivative_of_sigmoid(x):
    "Derivative of a sigmoid (analytical)"
    return sigmoid(x) * (1 - sigmoid(x))

# input
x = torch.linspace(-10,10,100, requires_grad=True)

# derivative of a sigmoid
dx = derivative_of_sigmoid(x)

# PyTorch program that implements sigmoid
z = sigmoid(x)

# using PyTorch autodiff to compute the derivative of the sigmoid
z_ = torch.sum(z)   # because backward can only be called on scalers
z_.backward()       # the backward pass

plt.figure(figsize=(8,8))
plt.title('Using PyTorch to compute the derivative of a sigmoid')
plt.plot(x.detach().numpy(), z.detach().numpy(), 'k', label='sigmoid')
plt.grid()
plt.plot(x.detach().numpy(), dx.detach().numpy(), 'b.', label='derivative computed analytically')
plt.plot(x.detach().numpy(), x.grad.detach().numpy(), 'r', label='derivative using autodiff')
plt.xlabel('x')
plt.legend();
```

Regularization for deep networks

Regularization: any modification to reduce generalization error but not the training errors:

- extra constraints and penalties
- prior knowledge

Deep learning is applied to extremely complex tasks. Consequently, regularization is not as simple as controlling the number of parameters

Regularization for deep networks

- Parameter norm penalties
- Data augmentation
  - Fake data
  - Successful in classification/object recognition tasks
- Noise injection
  - Applying random noise to the inputs
  - Applying random noise to hidden layers' inputs
    - Data augmentation at multiple levels of abstraction
  - Data augmentation almost always improves the performance of a neural network
  - Noise added to the weights
    - Recurrent neural networks
    - A practical stochastic implementation of Bayesian inference over weights
  - Noise can also bve added to target outputs

Summary

- Different ways to interpret a neural network
  - Compositions of non-linear functions
  - Computational graphs
  - Comprised of differentiable layers
    - Where possible compose new networks using existing networks

Summary

- Backpropagation: strategy for computing gradients for gradient-based learning
  - Use autodiff to automatixally compute gradients for each layer
    - Vast number of "deep learning" frameworks (e.g., TensorFlow, Theano, PyTorch, etc.); start with those first

Summary

- Controlling model complexity
- Deep learning
  - Loosely speaking, neural networks with several hidden layers
    - Convolutional layers: used for image processing
    - Fully connected layers: often used at the end for regression or classification

Readings

- Ch. 6-9 of Deep Learning by I. Goodfellow et al.

Copyright and License