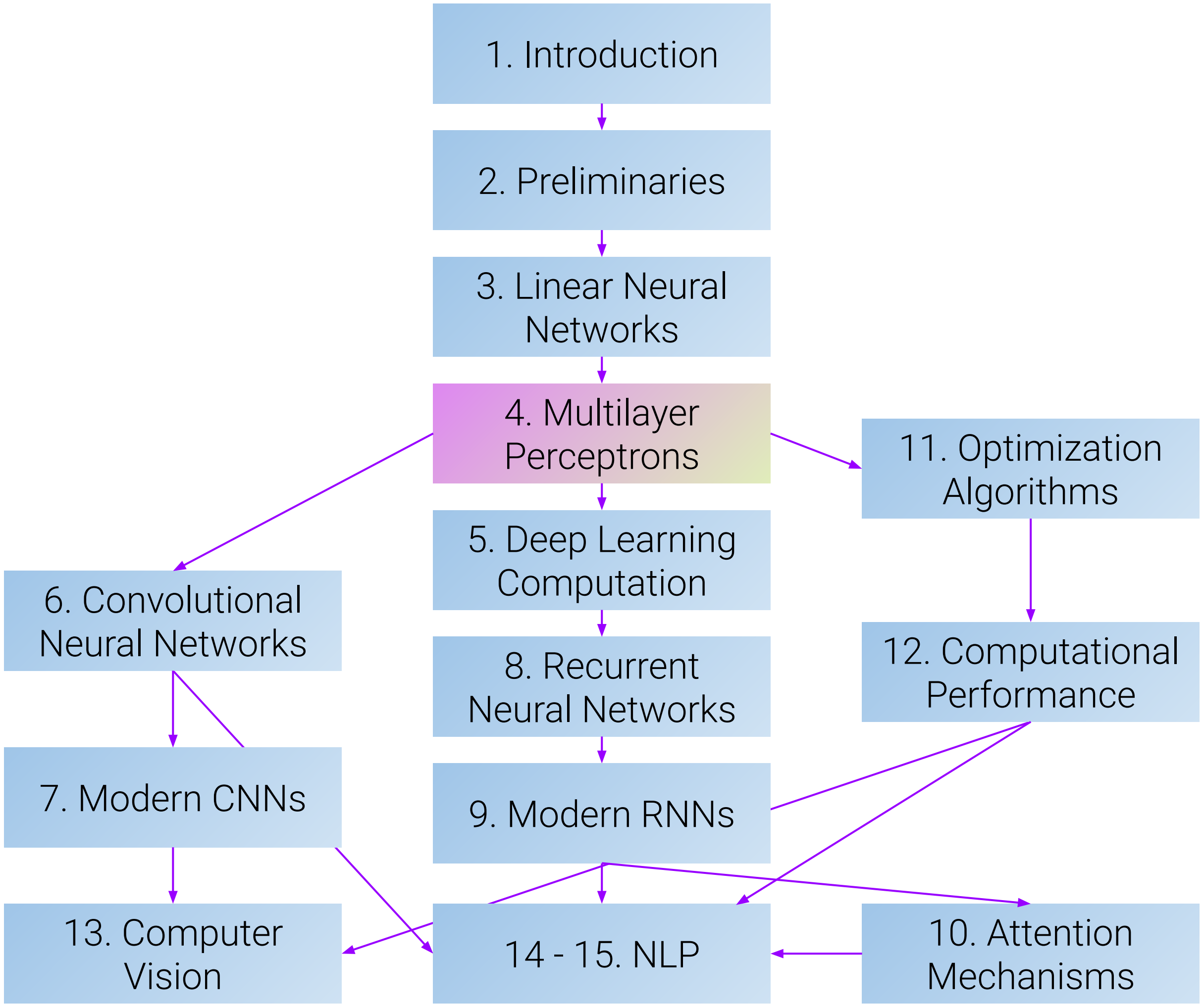
Multilayer Perceptrons

Session #4

A study group by dair.ai

Chapter 3


1. Introduction
2. Preliminaries
3. Linear Neural Networks
4. Multilayer Perceptrons
5. Deep Learning Computation
6. Convolutional Neural Networks
7. Modern CNNs
8. Recurrent Neural Networks
9. Modern RNNs
10. Attention Mechanisms
11. Optimization Algorithms
12. Computational Performance
13. Computer Vision
14 - 15. NLP

Agenda

- Multilayer perceptrons
- Implementation of multilayer perceptrons from scratch
- Model selection, underfitting, overfitting
- Regularization
  - Weight decay
  - Dropout
- Forward and Backward propagation
- Numerical stability and Initialization
- Environment and Distribution Shift
- Predicting house price on Kaggle

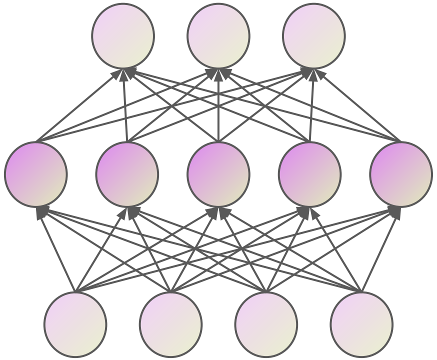
Linearity Assumption

- Linearity implies weaker assumption of ***monotonicity***
  - Increase in features cause
    - Increase in model's output (if weight is positive)
    - Decrease in model's output (if weight is negative)
- E.g., Increasing pixel intensity helps to distinguish between cats and dogs images
  - Inverted images may fail completely
  - Need more robust representations that consider context and relevant feature interactions
- We need a way to model more complex feature relationships

jointly learn a representation (via hidden layer) and a linear predictor acting on that representation

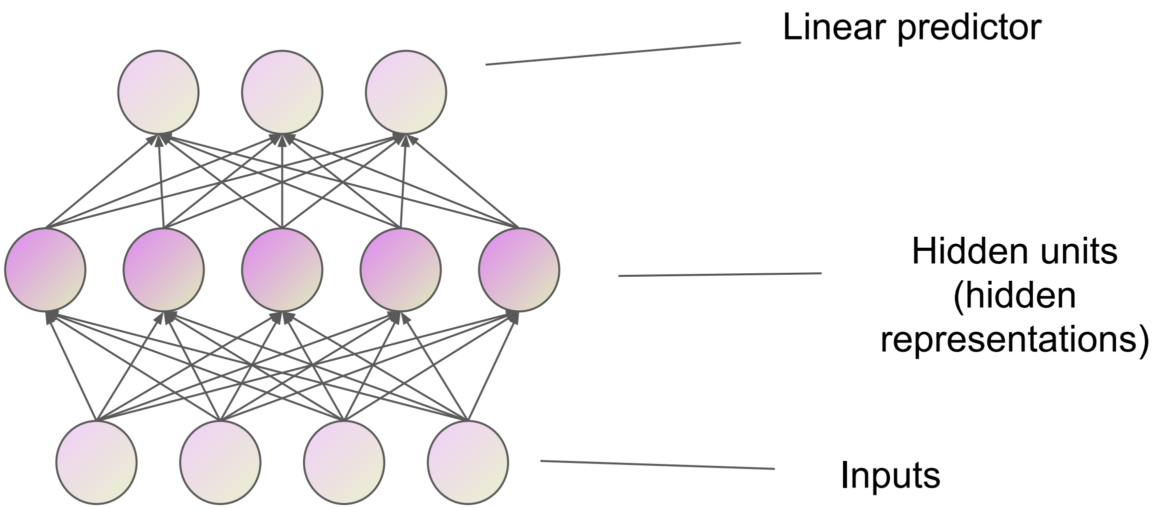
Multilayer Perceptron (MLP)

- Simple deep neural networks are called multilayer perceptrons (MLPs)
- ***Multiple layers*** of neurons fully connected
  - Hidden layers that learn representations
- Train high-capacity models that can help with ***overfitting***



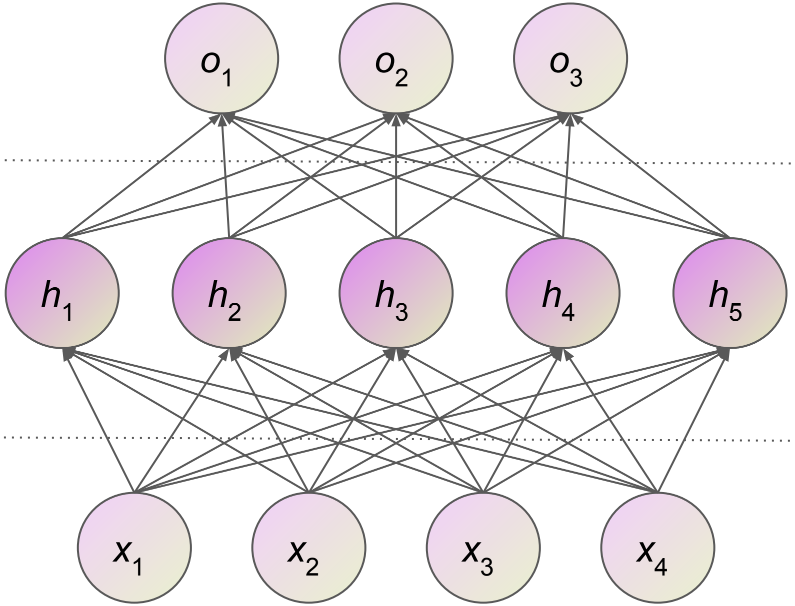
MLP Architecture

We can overcome limitations of linear models and handle more general functions by adding ***hidden layers*** in an MLP architecture



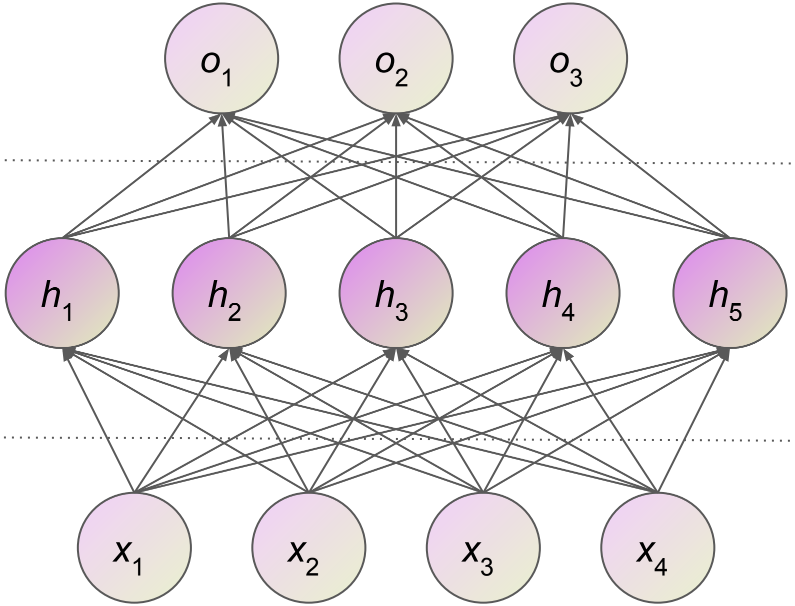
2-layer MLP (fully connected) - (Figure reproduced from d2l.ai)

MLP Architecture (Vanilla)



Output layer	$\mathbf{O} \in \mathbb{R}^{n \times q}$	$\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$	$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$
Hidden layer	$\mathbf{H} \in \mathbb{R}^{n \times h}$	$\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$	$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$
Input layer	$\mathbf{X} \in \mathbb{R}^{n \times d}$		

MLP Architecture (with Activation Function)



Output layer	$\mathbf{O} \in \mathbb{R}^{n\times q}$	$\mathbf{W}^{(2)} \in \mathbb{R}^{h\times q}$ $\mathbf{b}^{(2)} \in \mathbb{R}^{1\times q}$	$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$
Hidden layer	$\mathbf{H} \in \mathbb{R}^{n\times h}$	$\mathbf{W}^{(1)} \in \mathbb{R}^{d\times h}$ $\mathbf{b}^{(1)} \in \mathbb{R}^{1\times h}$	$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$
Input layer	$\mathbf{X} \in \mathbb{R}^{n\times d}$		

Activation functions

- Decide what neurons will be activated (control the flow of information)
- They are differentiable operators
- Adds ***non-linearity*** and allows the model to learn complex functions to fit data
- A few activation functions:
    - **ReLU** - produced derivatives that are well behaved and help mitigate *vanishing gradients*
    - **Sigmoid** - squashing function → (0, 1); used in binary classification problems
- Proper selection of activation functions become important for ***training stable and effective models*** (upcoming topic)

ML Generalization

- Memorization vs. generalization
- When training models on finite datasets there is risk of ***memorization***
  - The discovered associations appear to hold but they don't on unseen examples
  - Fail to fit the underlying distribution
  - Leads to ***overfitting***
- We desire to train a ***generalizable model*** that discovers a general pattern
  - Useful for making predictions on future unseen samples
  - Achieved by employing ***regularization*** techniques that tackle overfitting
- The ***training error*** is the error of the model on the training dataset
- The ***generalization error*** is expectation of the model's error
  - We estimate this error on independent ***test set*** from same underlying data distribution

Factors that affect model generalizability

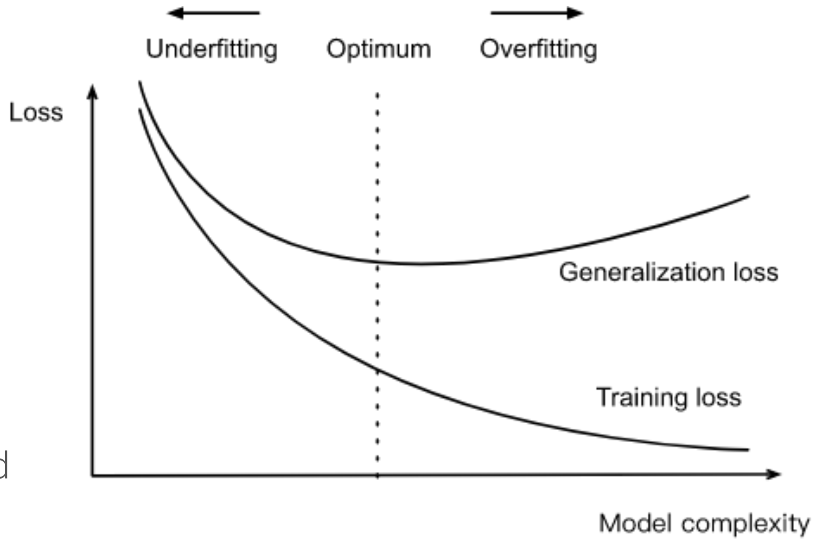
- **Tunable parameters**
  - Large number (too many features) could lead to overfitting (model complexity)
- **Parameter values**
  - Wider range of values can lead to overfitting (model complexity)
- **Training examples**
  - More data usually helps in tackling overfitting especially for deep neural networks

Model selection

- Models can differ in architecture, family, hyperparameters, etc.
- To avoid overfitting on test data, we use the ***validation dataset*** to determine hyperparameters and eventually select best model
- Compare different models on validation dataset
  - We are interested in generalization error
- In production, this gets more difficult
- When data is scarce, ***K-fold cross validation*** is employed

Underfitting and Overfitting

- **Underfitting**
  - Occurs when training and validation error are substantial and there is a small gap between them
  - Occurs when working with a simple model
  - A more complex model can help
- **Overfitting**
  - happens when training error is significantly lower than validation error
  - Regularization techniques help
  - Increasing # of examples
  - Model too complex (flexible and high capacity) and not enough data



Datasets and deep learning

- When working with small datasets, simpler models can perform better
- Deep learning thrives on large datasets
- However, large datasets can be expensive to collect or time consuming
- Assume we have good quality dataset, we can rely on regularization techniques to avoid overfitting:
    - Weight decay
    - Dropout

Weight Decay

- We can always reduce overfitting of the model by collecting more data
  - … but sometimes that process it too expensive or time consuming
- We can also address overfitting through a regularization technique known as weight decay
  - Commonly called L2 regularization and widely used in parametric ML models
  - Measure complexity of linear function
    - We want to ensure that the weight vector doesn't grow too large
    - In other words, we would like to keep a small weight vector (smaller weight values)
  - We add a norm of the weight vector as penalty term to original loss function and minimize both
  - Larger weight vector means that learning algorithm should focus on minimizing weight norm

$$L(\mathbf{w}, b) + \frac{\lambda}{2}||\mathbf{w}||^2$$

L2 vs. L1 Norm

- L1 norm emphasizes on models that concentrate weights on a small set of features while minimizing the effect of others (feature selection)
- L2 norm is computationally convenient and easy to obtain it's derivatives
- L2 norm emphasizes that the model distribute weight evenly across large number of features
  - By penalizing large components of the weight vector
  - Makes model more robust to measurement error in single variable?
- Weight update also considers shrinking of size of weight vector towards zero:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)$$

Dropout

- So far we used techniques of regularization that measure simplicity and reduce model complexity:
    - Tweaking degree of fitted polynomial (limiting number of features)
    - Weight decay (L2 regularization)
- Simplicity can also be represented through ***smoothness***
    - Neurons become less sensitive to the activation of another specific neuron
- Dropout enforces smoothness by injecting noise into each layer
    - Drops out some neurons during training by zeroing out fraction of nodes
    - Done on forward propagation and backward propagation
- Dropout break co-adaptation that happens between layers
    - Overfitting happens in a state where each layer depends on specific pattern of activation from previous layers (co-adaptation) → leads to overfitting

Dropout

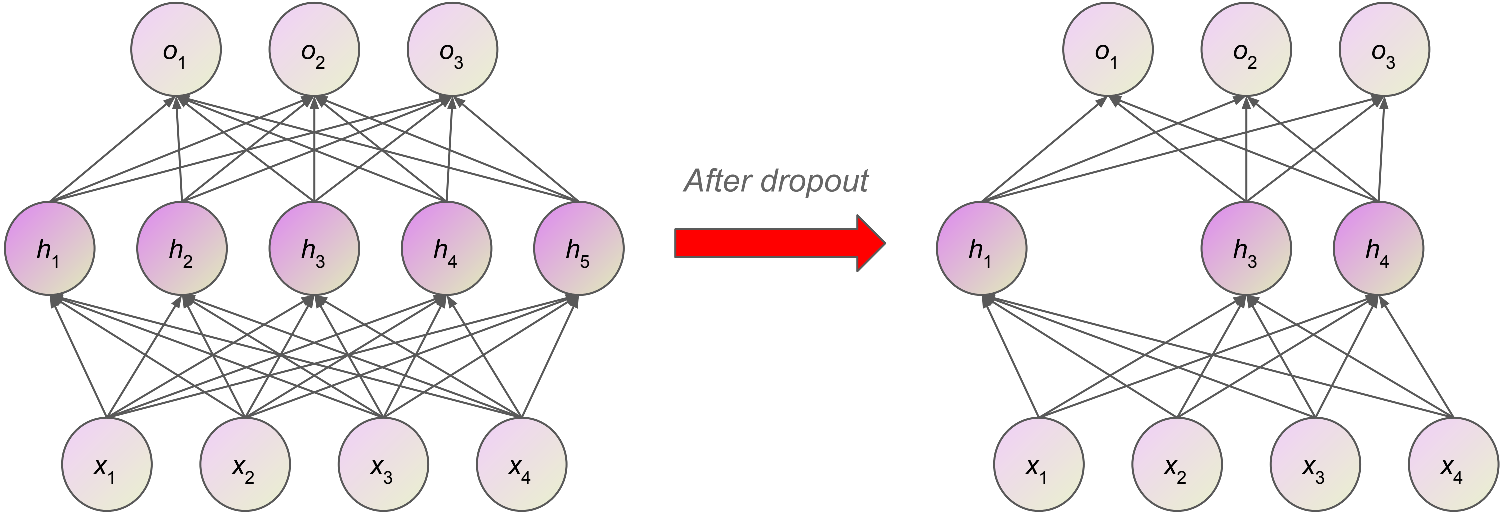
$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

- Dropout applied by injecting noise through debiasing of each layer
- Debiasing of layer achieved by normalizing using fraction of ***nodes retained***
- Apply dropout by zeroing out hidden unit with probability ***p***
- Intermediate activation replaced by random variable, ***h'***, as follows

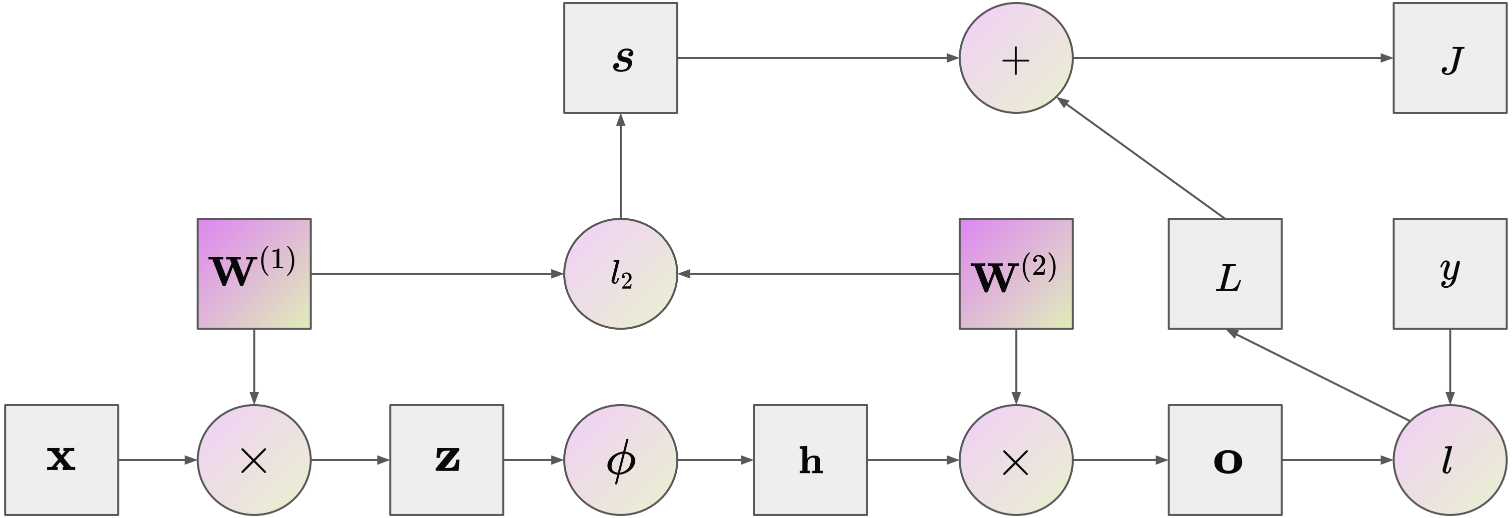
```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return torch.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = (torch.Tensor(X.shape).uniform_(0, 1) >
dropout).float()
    print(mask)
    return mask * X / (1.0 - dropout)
```

Dropout intuition

- Reduces the over dependence of output on elements in the hidden layer
- Note that their respective gradients also vanish during backpropagation

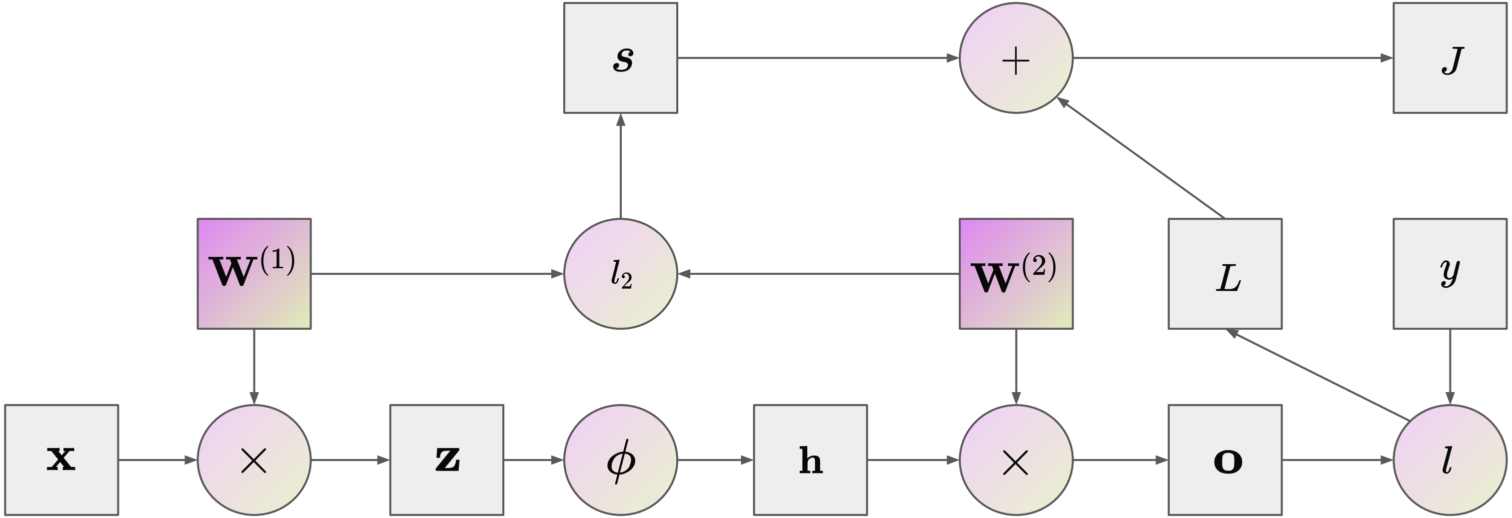


Forward Propagation



Forward propagation (Figure reproduced from d2l.ai)

Backward propagation (Assignment 3)



Numerical stability and Initialization

- **Initialization plays a huge role in training:**
  - Maintaining numerical stability and ensure parameters & gradients remain well controlled
  - Different initialization schemes can be combined with activation functions in interesting ways
  - Functions and initialization impact how fast optimization algorithm converges
  - If not done right, it may lead to vanishing and exploding gradients
- **Unstable gradients lead to:**
  - Unstable optimization algorithm
  - Parameter updates that are too large… and destroy the model
  - Parameter updates that are too small… and render learning impossible

Environment and Distribution Shift

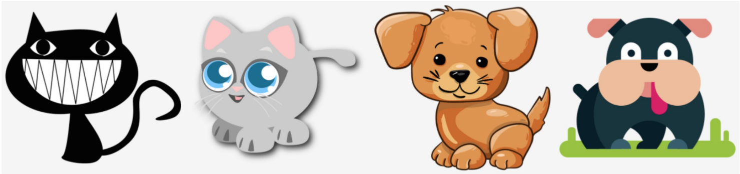
- As ML practitioners, it's important to understand ***origin of data*** and what we plan to do with ***model outputs***
- Distribution shift in data can lead to failed models in production
- Users of your ML models may obtain insights/knowledge into how ML model produce outputs and use it to their advantage (e.g., loan repayment criteria)
- Some concerns:
  - Simple models
  - Models that are too technically difficult
  - How about ethical use of algorithms?

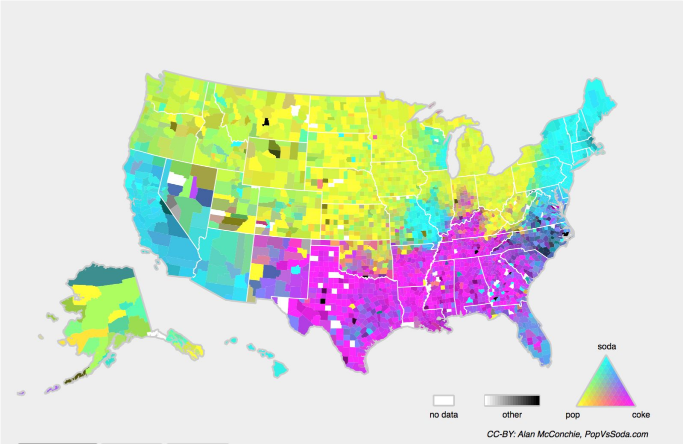
Type of Distribution shift

- **Covariate shift**
  - A shift in the distribution of the features
  - Training on realistics photos, testing on cartoons
- **Label shift**
  - A shift in the distribution of labels
  - Predicting diagnoses based on symptoms
    - Disease cause symptoms
- **Concept shift**
  - Definitions of labels change (jobs, mental illness, soft drink name in the US)
  - Typically occurs temporally or geographically

cat cat dog dog

cat cat dog dog





A taxonomy of learning problems

- **Batch learning**
  - Train once, deploy and never update again (app: only let cats into the house)
- **Online learning**
  - Continuously improve learning algorithm based on observations (app: stock price prediction)
- **Bandits**
  - Limit number of actions that can be taken
- **Control**
  - Use automatic control theory (e.g., improve diversity of generated text)
- **Reinforcement learning**
  - Support better interaction with robust environment (e.g., Go, Chess, self-driving car)

Announcements

- I will post a recording of chapter 5 in the coming days
- Next week we will have Samil delivering CNNs and modern CNNs

Salim Chemlal, Ph.D.
Deep Learning Instructor at NVIDIA