Introduction

2021年9月7日 7:34

Machine learning: create new functions using example behavior rather than explicit instructions

- Approximate functions (not perfect, but accurate enough) that can be applied to new data

Deep learning: specific type of ML using neural networks
ML and Deep learning are a type of AI

Labelled data:

- Examples come with the expected answer
- It provides an example of an input to output mapping from which we would like the ML system to generalize for other similar inputs
    - There might be overfit

What problems for ML

- Lots of high-quality data is available
- desired output is clear, unambiguous and testable
- the input to output relationship is not already well understood

Things that don't need ML:

- Clear and well understood mathematical relationship between input and output
- Clear and well understood physical relationship between the input and output
    - Trajectory formula
    - Learn the gravitational function
- Clear and well understood algorithmic relationship between the input and output

Deep learning can't: why, explain, plan, deductive reasoning, design

- Deep learning provides answers, but not justifications

Neural networks

- Know how to train them efficiently
- Back propagation quickly and efficiently find a high quality approximate function
    - Basis of success for neural networks'

Deep neural networks

- Networks with many trainable layers, which allows them to express very complex functions
- Generally effective when we have a very large set of training data

Where does deep learning work well

- Problems where the input is unstructured data
    - Images/video, natural language
- Problems with complex relationships but clear goals
    - Classifying images
    - Identifying objects

AI and AGI

- AI (Artificial Intelligence): any technique that makes computers act intelligently
- AGI (Artificial General Intelligence): making computers smart like us

Data science: process of using data analysis to build understanding
Machine learning: process of using example data to create approximate functions that can then be applied to new data. (understanding is rarely provided)
Neural networks: ML using an interconnected network of trainable artificial neurons (perceptrons) that maps some input to an output

Deep learning: ML using multi-layered neural networks, which are normally trained with large data sets
Supervised learning: ML when the example data provides both the expected input and output. You can supervise the training process by identifying and correcting mistakes
Labelled data: example data that includes the expected output, used in supervised learning
Unsupervised learning: ML when only expected input is provided. In this case, the ML system learns relationships between the inputs themselves.
Unlabeled data: example data that does not include the expected output, used in unsupervised learning.
Reinforcement learning: ML which uses only high-level goals and repeated trial and error during training

Machine learning and logistic regression

September 14, 2021 1:31 PM

Fundamental challenge of ML: the machine can only learn if we have examples that we can use to train it.

Logistic regression

- It is a technique that assumes that we can make a prediction (hypothesis) bases on a linear combination of the inputs
  - $z = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n + b = w^T X + b$.
  - w and b are called parameters, we want to find the correct parameters

Binary classification

- Classify data into 2 groups
- Can use 0 and 1 to represent each
- Sigmoid: a function that forces values between 0 and 1
  - $\sigma(x) = \frac{1}{1+e^{-x}}$.

Final logistic equation:

- $a = \sigma(w^T X + b)$.
- If $a > 0.5$, we predict 1
- If $a \leq 0.5$, we predict 0

To find parameters:

- Guess and test
- Simulated annealing
- Genetic algorithms
- Gradient descent

Cost function (J)

- A way to compare combinations of w and b to know which works best
- It is a measure of fitness of any given selection of w and b.
- If $J(w_1', w_2', \ldots, w_n', b') < J(w_1, w_2, \ldots, w_n, b)$, then w', b is a better set of parameters selection than w, b.
- First solution: accuracy = right answers/total answers.
- Parameter adjustment
  - If $J(w_1, \ldots, b)$ is the overall cost, then $\frac{\partial J}{\partial w_1}$ is the rate of change of the cost w.r.t w_1.
  - Then we can improve the parameters by:
    - $w = w - \alpha \frac{\partial J}{\partial w}$.
    - $b = b - \alpha \frac{\partial J}{\partial b}$.
    - Learning rate: α is the size of the adjustment
- Building a cost function
  - It needs to be differentiable, convex function
  - When $y = 1$: $L(a, y) = -\log(a)$
  - When $y = 0$: $L(a, y) = -\log(1 - a)$
  - We can combine them: $L(a, y) = -(\log(a) + \log(1 - a))$.
  - Using chain rule, we can find that $\frac{\partial L}{\partial w_n} = x_n(a - y), \frac{\partial L}{\partial b} = a - y$.
  - Finally, $J = -\frac{1}{m}\left(\sum_{i=1}^{m} y^i \log(a^{(i)}) + \sum_{i=1}^{m}(1 - y^i)\log(1 - a^{(i)})\right)$.
    - $\frac{\partial J}{\partial w_n} = \frac{1}{m}\sum_{i=1}^{m} x_n^i (a^i - y^i)$.
    - $\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^i - y^i)$.

Main algorithm

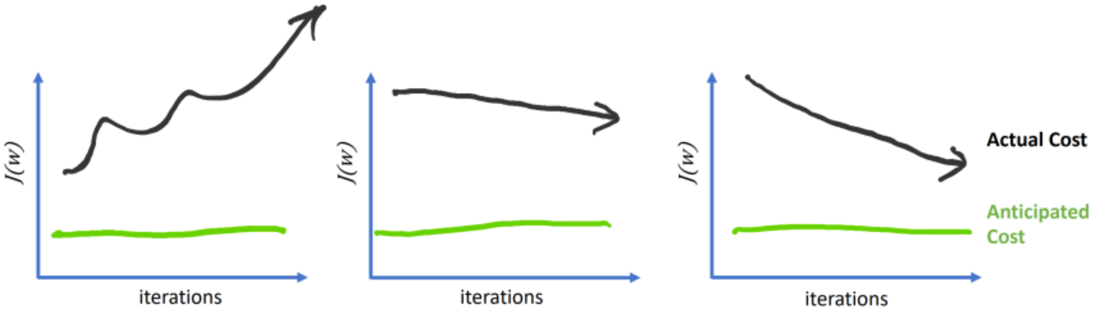
- Assume: $a = \sigma\left(w^T X + b\right)$.
- Initialize w, b to random values or zero
- Repeatedly apply: $w = w - \alpha \frac{\partial J(w,b)}{\partial w}$, $b = b - \alpha \frac{\partial J(w,b)}{\partial b}$.
- Stop when $J <$target error

The goal is prediction, it only matters if it works for new data

Sources of inaccuracy

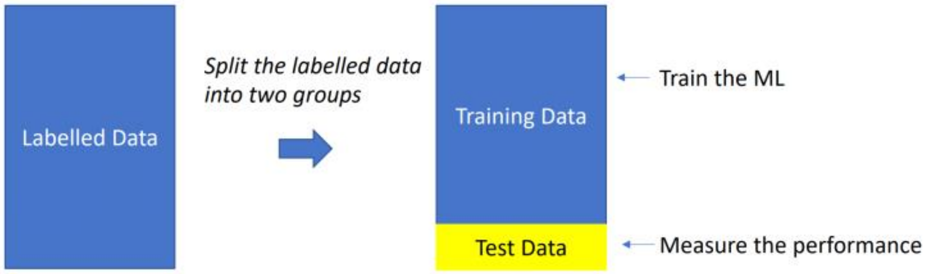
- AI model does not match the underlying nature of the data (data is not linearly separable)
- Learning algorithm did not find the best set of parameters for the model
- The example data is not representative of the new data
  - Not enough data to represent function
  - The data is noisy
  - The underlying behavior is not deterministic

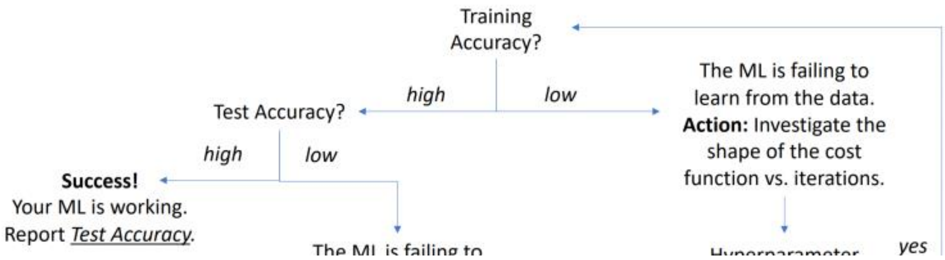
Hyperparameters in logistic regression

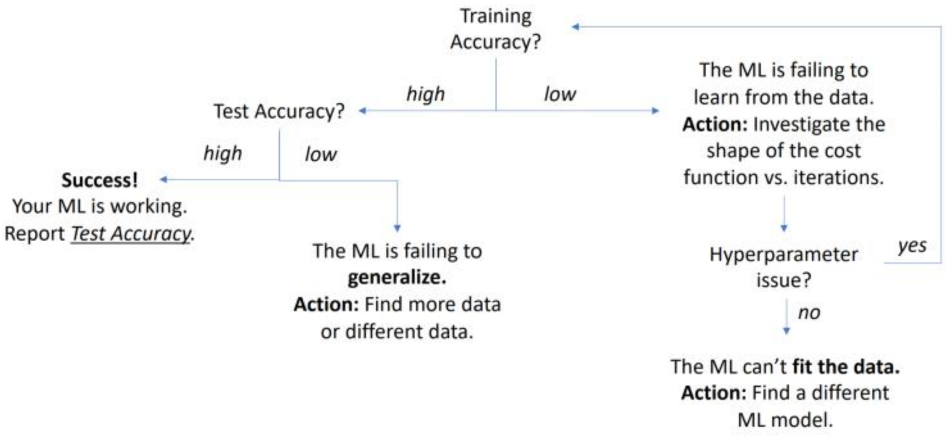
- Learning rate α:
  - Too large: final parameters are worse than random
  - Too small: final parameters are better than random, but not optimal
- Number of iterations
  - Too small: final parameters are better, but not optimal
  - Too large: as long as the learning rate is small enough, this only costs CPU cycles
- 

Learning rate is too **high**. Learning rate is too **low**. Number of iterations to **low**.

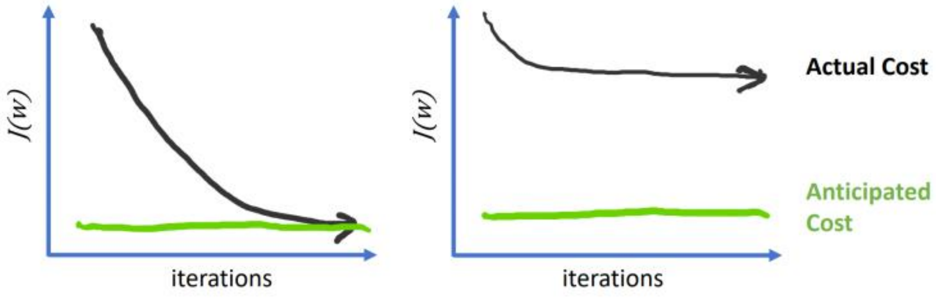
Build a test data set

- Most important: take some of the data and put it off to the side
- 

- 

- 
Training Accuracy?
high
low
Test Accuracy?
high
low
Success!
Your ML is working.
Report Test Accuracy.
The ML is failing to generalize.
Action: Find more data or different data.
The ML is failing to learn from the data.
Action: Investigate the shape of the cost function vs. iterations.
Hyperparameter issue?
yes
no
The ML can't fit the data.
Action: Find a different ML model.

Model Issues



Good behavior.

Learning unsuccessful.
(Data/model fit issue)

Different issues

- AI model doesn't fit data
  - Training accuracy is low and hyperparameter tuning doesn't help
  - Consider a different AI model
- We are not finding the best parameters
  - Unexpected shape of cost/iterations graph
  - Tune the hyperparameters
- Example data does not represent the new data (lack of data, noisy data, non-deterministic data)
  - High training accuracy but low test accuracy
  - Try to find more, better or different data

Reporting the accuracy of the ML system

- Select a representative test data set from the labelled data
- Make sure we don't use the test data to train the ML
- Report the accuracy of the test data set

Vectorization

- Machine learning are computationally expensive
  - Best solutions comes from:
    - A lot of example data
    - Models that contain a lot of parameters

    - Trained over a lot of iterations
  - It is critical to find high quality solutions in a reasonable timeframe
-

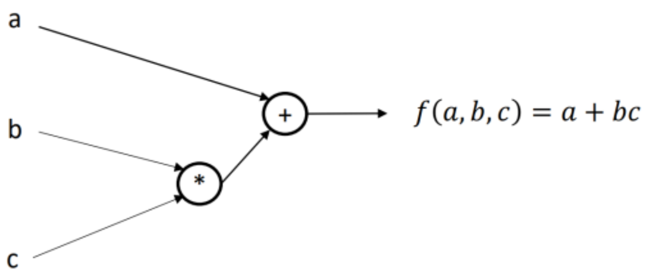
Neural network

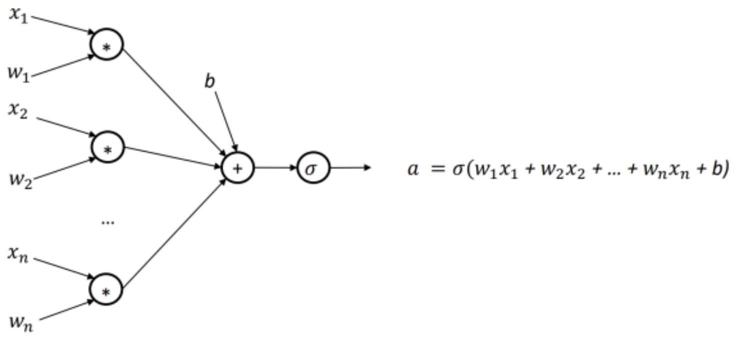
September 21, 2021 1:22 PM

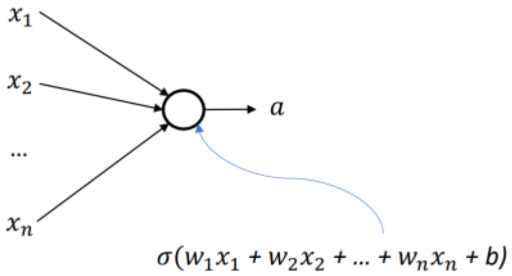
Problem with logistic regression:

- The assumption about linear relationship
- We can continue to add various terms to logistic regression and gradient descent will work

Neural networks

- Can learn very complex non-linear relationships between an arbitrary number of features across an arbitrary number of examples rather than having to specify them
- It is a type of computation graph inspired by an idealized view of a real neuron
- Computation graphs:
    - A way to specify a computation relationship between inputs and outputs
    - 

    - Logistic regression
        - 

        - Shorthand:
        - 

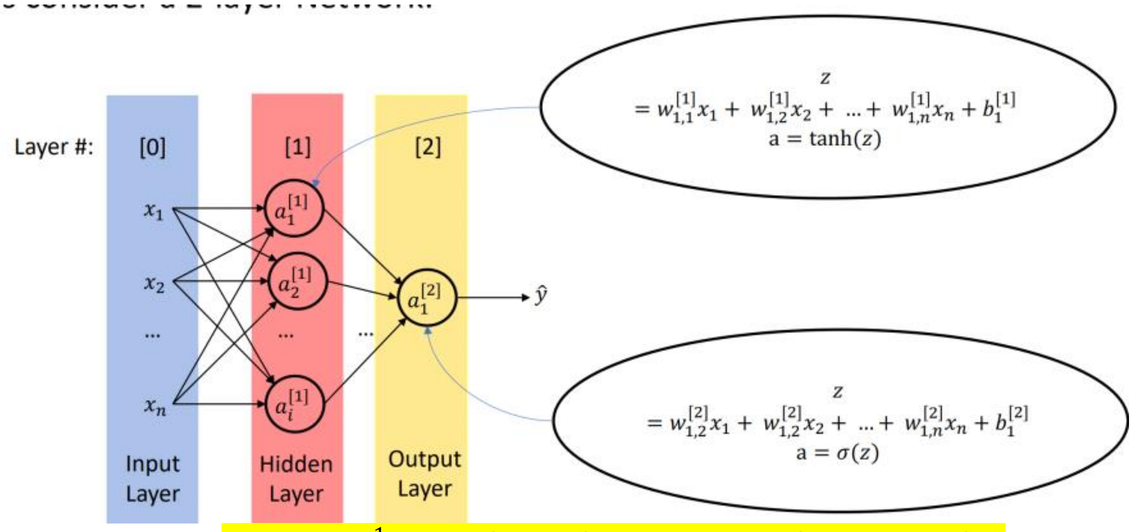
        - This is a small neural network
- We use *tanh* rather than sigmoid in the middle layers for neural networks
- Emergent behavior: connecting even a small number of units with simple behaviors enables the approximation of very complex functions
- However, it can be trained in a straight-forward and efficient manner

Activation function

- Connecting multiple linear regression units does not add much new flexibility
- The non-linear activation function (like tanh and sigmoid) is the key to allowing combinations of logistic regression units to produce complex functions

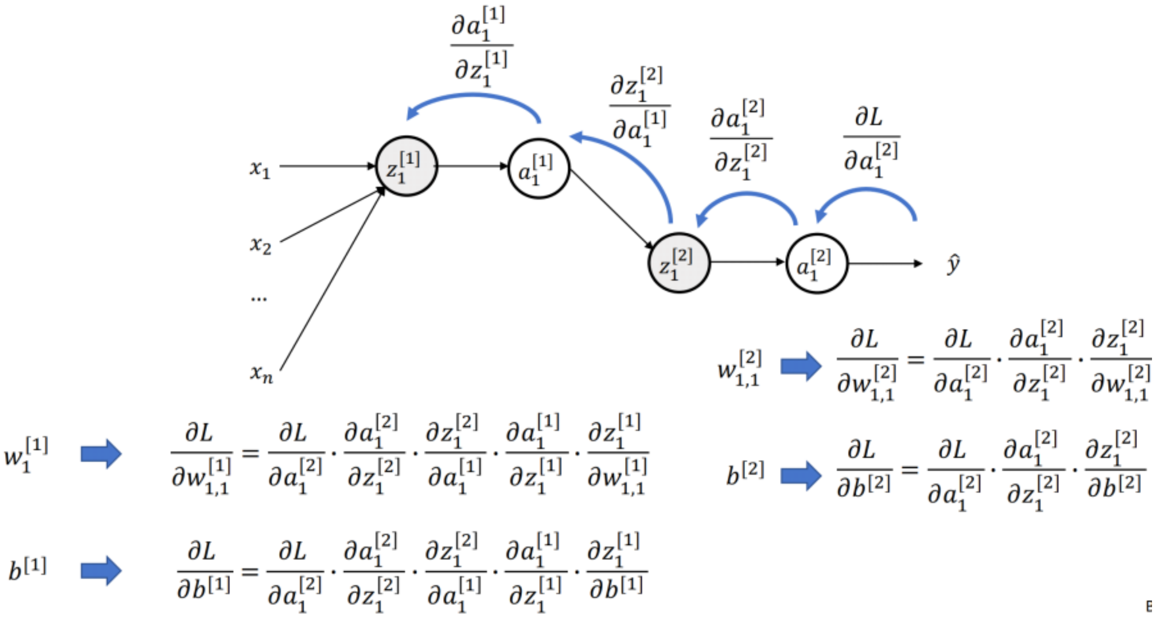
- The parameter directly effects the location of the decision boundary
- Without it, all combinations of logistic regression would continue to be linear

Formalization



$z = w^{[1]}_{1,1}x_1 + w^{[1]}_{1,2}x_2 + \ldots + w^{[1]}_{1,n}x_n + b^{[1]}_1$
$a = \tanh(z)$
Layer #:
[0]
[1]
[2]
x_1
x_2
...
x_n
$a^{[1]}_1$
$a^{[1]}_2$
...
$a^{[1]}_i$
...
$a^{[2]}_1$
$\hat{y}$
$z = w^{[2]}_{1,2}x_1 + w^{[2]}_{1,2}x_2 + \ldots + w^{[2]}_{1,n}x_n + b^{[2]}_1$
$a = \sigma(z)$
Input Layer
Hidden Layer
Output Layer

- Cost function: $J(\hat{y},y) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^i \log\left(\hat{y}^{(i)}\right) + \sum_{i=1}^{m}\left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right)$.
  - Use $\hat{y}$ to denote the activation of the output layer of the NN.
- Back propagation
  - Step 1: calculate $\hat{y}$ using computation graph
  - Step 2: determine the loss
  - Step 3: update each parameter (using the partial derivative of cost)



$\frac{\partial a^{[1]}_1}{\partial z^{[1]}_1}$
$\frac{\partial z^{[2]}_1}{\partial a^{[1]}_1}$
$\frac{\partial a^{[2]}_1}{\partial z^{[2]}_1}$
$\frac{\partial L}{\partial a^{[2]}_1}$
x_1
$z^{[1]}_1$
$a^{[1]}_1$
x_2
...
x_n
$z^{[2]}_1$
$a^{[2]}_1$
$\hat{y}$

$$w^{[2]}_{1,1} \Rightarrow \frac{\partial L}{\partial w^{[2]}_{1,1}} = \frac{\partial L}{\partial a^{[2]}_1} \cdot \frac{\partial a^{[2]}_1}{\partial z^{[2]}_1} \cdot \frac{\partial z^{[2]}_1}{\partial w^{[2]}_{1,1}}$$

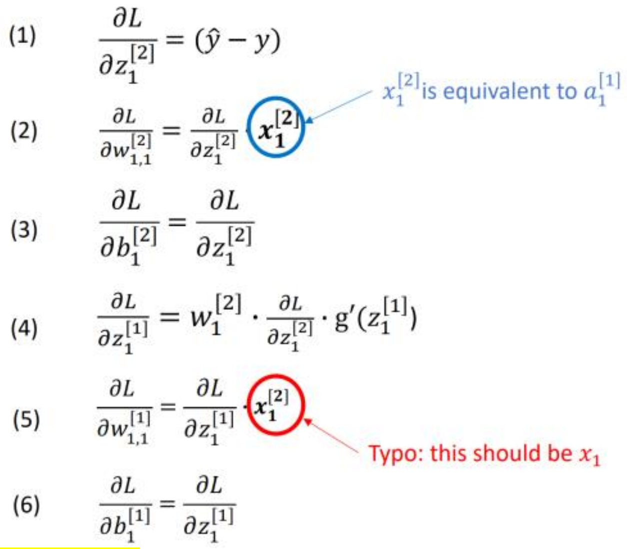
$$w^{[1]}_1 \Rightarrow \frac{\partial L}{\partial w^{[1]}_{1,1}} = \frac{\partial L}{\partial a^{[2]}_1} \cdot \frac{\partial a^{[2]}_1}{\partial z^{[2]}_1} \cdot \frac{\partial z^{[2]}_1}{\partial a^{[1]}_1} \cdot \frac{\partial a^{[1]}_1}{\partial z^{[1]}_1} \cdot \frac{\partial z^{[1]}_1}{\partial w^{[1]}_{1,1}}$$

$$b^{[2]} \Rightarrow \frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial a^{[2]}_1} \cdot \frac{\partial a^{[2]}_1}{\partial z^{[2]}_1} \cdot \frac{\partial z^{[2]}_1}{\partial b^{[2]}}$$

$$b^{[1]} \Rightarrow \frac{\partial L}{\partial b^{[1]}} = \frac{\partial L}{\partial a^{[2]}_1} \cdot \frac{\partial a^{[2]}_1}{\partial z^{[2]}_1} \cdot \frac{\partial z^{[2]}_1}{\partial a^{[1]}_1} \cdot \frac{\partial a^{[1]}_1}{\partial z^{[1]}_1} \cdot \frac{\partial z^{[1]}_1}{\partial b^{[1]}}$$

    - The same derivatives are re-used across and back through the NN
- The logistic regression gives the last layer in the NN
  - $\frac{\partial L}{\partial a^{[2]}_1} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$.
  - $\frac{\partial a^{[2]}_1}{\partial z^{[2]}_1} = \hat{y}(1-\hat{y})$.
  - $\frac{\partial z^{[2]}_1}{\partial w^{[2]}_{1,1}} = x^{[2]}_1, \ldots, \frac{\partial z^{[2]}_1}{\partial b^{[2]}} = 1$.
- The cross NN layers:
  - Consider the equations:
    - $z^{[2]}_1 = w^{[2]}_{1,1}a^{[1]}_1 + w^{[2]}_{1,2}a^{[1]}_2 + \cdots + w^{[2]}_{1,n}a^{[1]}_n + b^{[2]}_1$, so $\frac{\partial z^{[2]}_1}{\partial a^{[1]}_1} = w^{[2]}_{1,1}$.
    - $a^{[1]}_1 = \tanh z^{[1]}_1$, so $\frac{\partial a^{[1]}_1}{\partial z^{[1]}_1} = 1 - \tanh^2 z^{[1]}_1$
  - So:

- $\frac{\partial L}{\partial z_1^{[2]}} = \hat{y} - y$.
- $\frac{\partial L}{\partial z_1^{[1]}} = w_{1,1}^{[2]}(\hat{y} - y)\left(1 - \tanh^2\left(z_1^{[1]}\right)\right)$.

- 

  (1) $$\frac{\partial L}{\partial z_1^{[2]}} = (\hat{y} - y)$$

  (2) $$\frac{\partial L}{\partial w_{1,1}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \, x_1^{[2]}$$

  $x_1^{[2]}$is equivalent to $a_1^{[1]}$

  (3) $$\frac{\partial L}{\partial b_1^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}}$$

  (4) $$\frac{\partial L}{\partial z_1^{[1]}} = w_1^{[2]} \cdot \frac{\partial L}{\partial z_1^{[2]}} \cdot g'(z_1^{[1]})$$

  (5) $$\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \cdot x_1^{[2]}$$

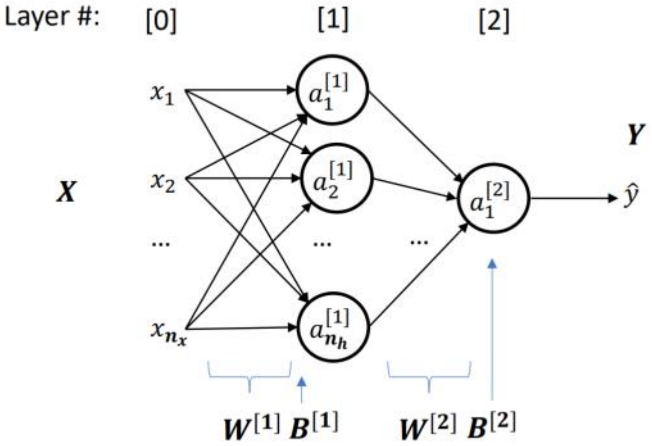
  Typo: this should be x_1

  (6) $$\frac{\partial L}{\partial b_1^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}}$$

- Interpretation
  - We are propagating the error and attributing it to each node and then each parameter
  - When $\hat{y} - y \approx 0$, none of the parameters are adjusted

Implementation

- Use vectorization to group operations together
- Avoid re-calculating values that are used repeatedly
- Number of layers and number of neurons in each layer are hyperparameters
- For 1 hidden layer and single output



  - n_x is the number of input features
  - n_h is the number of hidden units in layer [1]
  - $W^{[1]}$ is a matrix of all the parameters in layer[1] with shape (n_h, n_x).
  - $W^{[2]}$ is a matrix of all the parameters in layer[2] with shape $(1, n_h)$.
  - $B^{[1]}$ is a vector of the bias parameters in layer[1] with shape $(n_h, 1)$.
  - $B^{[2]}$ is a vector of the bias parameters in layer[2] with shape (1,1).
  - m is the number of examples in the training data set
  - X is a matrix of all the input features for all examples in the training data set with shape (n_x, m).
  - Y is the labels for all the examples in the training data set with shape $(1, m)$.
  - Forward propagation
    - Consider a single example i,
      - □ $z^{[1](i)} = W^{[1]}x^{(i)} + B^{[1]}$, $a^{[1](i)} = g\left(z^{[1](i)}\right)$, $g(z) = \tanh z$.
      - □ $z^{[2](i)} = W^{[2]}a^{[1](i)} + B^{[1]}$, $\hat{y} = a^{[2](i)} = \sigma\left(z^{[2](i)}\right)$.

        - For all m examples
            - $Z^{[1]} = W^{[1]}X + B^{[1]}, A^{[1]} = g\left(Z^{[1]}\right), g(Z) = \tanh Z.$
            - $Z^{[2]} = W^{[2]}A^{[1]} + B^{[1]}, \hat{Y} = A^{[2]} = \sigma\left(Z^{[2]}\right).$
    - Back propagation (vectorized)
        - $dZ^{[2]} = \hat{Y} - Y.$
        - $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}.$
        - $dB^{[2]} = \frac{1}{m} \sum dZ^{[2]}.$
        - $dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]}).$
        - $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}.$
        - $dB^{[1]} = \frac{1}{m} \sum dZ^{[1]}.$
    - Parameter update (Vectorized)
        - $W^{[1]} = W^{[1]} - \alpha dW^{[1]}.$
        - $B^{[1]} = B^{[1]} - \alpha dB^{[1]}.$
        - $W^{[2]} = W^{[2]} - \alpha dW^{[2]}.$
        - $B^{[2]} = B^{[2]} - \alpha dB^{[2]}.$
    - Repeat until cost < target
- Parameter initialization
    - Setting all parameters to 0 does not work
    - Uniform non-zero value does not work
    - The initialization should be random numbers

Hidden units/layers

- NN architecture is extremely flexible. We can define any number of hidden layers and any number of units per layer
- However, more units are not necessarily better (cost in terms of training and deployment computing resources/time)
- Extra units contribute to overfit

Overfit in NN

- The best answer is the one that is the most accurate on new data
- A learned solution that track too close to the training data risks missing the big picture and simply memorizing training data
- The number of hidden layers and the number of units/layer are hyperparameters to be tuned to achieve optimal performance

Validation

- For logistic regression, we need two data sets (test and training)
- For NNs, we need 3 data sets, because of the overfit
    - Training data: train the model
    - Validation data: tune the hyperparameters
    - Test data: measure the performance
- The validation data set gives us data that was not used to train the NN, but can be used to tune the hyperparameters
- The test data set then gives us independent reference to measure the performance of the AI

Images as input data

- Grayscale image
    - Can be modelled as an array of pixels
    - Each array value is $[0,255]$ representting brightness of the pixel.
    - 0 for black and 255 for white
- Color image
    - Model as three channels (RGB), $H \times W \times 3$.
    - Feature vector:
        - Flatten each array into a vector and concatenate

        - It becomes a vector of length $3HW$.
- Each pixel is a feature, can use LR and NN to classify

Multiclass classification

- Number of possible classes n_c.
    - $n_c = 2$ for the binary classification.
    - $n_c = 10$ for MNIST
    - $n_c = 10$ for CIFAR
    - $n_c = 20{,}000$ for Image Net
    - $n_c = 9$ for ISIC
- Versus multilabel
    - Multiclass: input has exactly one label
    - Multilabel: input has one or more labels
- Output encoding:
    - One-hot encoded vector of length n_c.
    - It maps discrete categories to single continuous output
    - It allows us to extend what we know about building binary classification models
- Common approaches
    - Multiple binary classifiers
        - One-vs-all (one-vs-rest)
            - □ Build n_c binary classifiers
            - □ One binary classifier per class
            - □ Each classifier predicts whether the input is in its class or not
            - □ Classes may overlap, sample may be in more than one or none of the classes
        - One-vs-one
            - □ Build $\frac{n_c(n_c-1)}{2}$ binary classifiers (all possible combinations of 2 classes)
            - □ Each classifier only receives data about the pair of classes it is discriminating between
            - □ Use a majority voting scheme to select the class that was predicted the most often among the binary classifiers
            - □ Scales poorly with number of classes
            - □ Performs about the same as One-vs-all
    - Single classifier with multiple outputs
        - Deep neural networks
        - Change output layer to have one node per class, each output continues to act as a binary classifier for that class
        - Has n_c output nodes
        - Classes are mutually exclusive

Activation function (softmax):

- It normalizes the output such that each output node continues to produce a value between 0 and 1.0 and also sum to 1.0
- Can interpret this as a set of prediction probabilities for each class
- Input: a vector Z of length n_c
- Function: $g_i(Z) = \frac{e^{z_i}}{\sum_{j=1}^{n_c} e^{z_j}}$.
- We finally choose the class with the highest probability
- It is a generalization of sigmoid

Categorical Cross Entropy Loss (Softmax Loss):

- Generalization of the Binary Cross Entropy Loss
- $L(\hat{y}, y) = -\sum_{j=1}^{n_c} y_j \log \hat{y}_j$.
    - For $n_c = 2$:
        - $L = -(y_1 \log \widehat{y_1} + y_2 \log \widehat{y_2})$.
        - $y_2 = 1 - y_1, \widehat{y_2} = 1 - \widehat{y_1}, \hat{y} = P(y = 1|x)$.
- It quantifies the difference between two probability distributions over the same underlying set

of events

    - A true distribution (true labels)
    - An estimated distribution (predicted labels)

Cost function

- Minimize the average loss across all training samples.
- $J(W,B) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$.

Back propagation

- $\frac{\partial L}{\partial z_1} = \widehat{y_1} - y_1$.
- $\frac{\partial L}{\partial z_2} = \widehat{y_2} - y_2$.
- $\frac{\partial L}{\partial z_n} = \widehat{y_n} - y_n$.

Summary of single neural network with multiple output

- One output node for each class
- Use Softmax activation on final layer
- Minimize the categorical cross-entropy loss
- Train on one-hot encoded label data
- Cannot be used for multi-label classification

Multilabel classification

- Cannot use softmax
- Use separate classifiers or use sigmoid on outputs
- Labels cannot be one-hot encoded vectors

Deep Neural Networks

September 27, 2021 12:24 PM

General points

- It is an extended version of 2-layer neural networks
- We count layers that have parameters
- Fully Connected (FC): each input connects to each node
    - Each FC layer can have different number of units
    - Also referred to as Multilevel Perceptron (MLP)
- Number of parameters per FC layer:
    - Weights: $n^{[l-1]} * n^{[l]}$.
    - Biases: $n^{[l]}$.

Layers and vectorized forward propagation

- Arranged in layers for vectorized computation
- Activation function is not required to be the same in the same layer

Increase capacity of the approximation function

- A neural network with one hidden layer provides the mapping:
    - $Y(X) = \sigma\left(W^{[2]} \tanh\left(W^{[1]}X + B^{[1]}\right) + B^{[2]}\right)$.
- This is a class of functions and each member function of this class is realized by a specific set of values for the parameters

Feature space transformation

- For $\tanh(Wx + b)$.
    - A linear transformation of W.
    - A translation of b
    - An application of $\tanh$.
- With logistic regression (any linear classifier), we can manually transform features to encode non-linearity
    - This is called feature engineering and requires analysis and human effort
    - Data then could be linearly separable

There is no formal definition of deep neural network
The number of layers does not matter too much

Universal approximation theorem

- A neural network with one hidden layer can approximate any continuous function
- But whether the suitable parameters can be found easily or how many units we need are unanswered
- In practice, deep networks generally perform better than shallow ones, especially on unstructured data with wide variation

Problems that deep learning works well

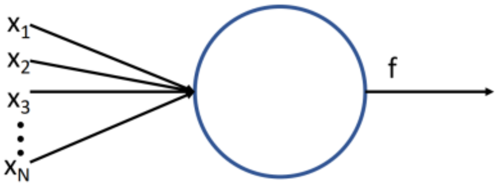
- Input is unstructured data
    - Images/video
    - Radar
    - X-ray
    - Audio/voice
    - Natural language
    - Mixed data
- Problems with complex relationships but clear goals
    - Classifying images
    - Identifying objects

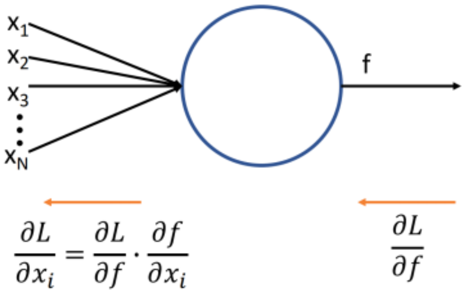
    - Winning chess
    - Predicting consumer behavior

Back propagation through softmax and categorical cross-entropy

- Consider $n_c = 3, L = -y_1 \log \widehat{y_1} - y_2 \log \widehat{y_2} - y_3 \log \widehat{y_3}$
    - $\frac{\partial \widehat{y_1}}{\partial z_1} = \widehat{y_1}(1 - \widehat{y_1})$.
    - $\frac{\partial \widehat{y_2}}{\partial z_1} = -\widehat{y_2}\widehat{y_1}$.
    - $\frac{\partial \widehat{y_3}}{\partial z_1} = -\widehat{y_3}\widehat{y_1}$.
    - $\frac{\partial L}{\partial z_1} = \widehat{y_1} - y_1$.
    - $\frac{\partial L}{\partial z_2} = \widehat{y_2} - y_2$.
    - $\frac{\partial L}{\partial z_3} = \widehat{y_3} - y_3$.

Back Propagation on computation graphs

- Calculating closed-form partial derivatives become infeasible and error prone with deep networks and many parameters
- If we want to try a different loss function or make architectural changes like trying different activation functions, need to derive again
- At graph construction
    - Assign variable names to each intermediate node's output
    - Re-express each node as a function of its immediate inputs
    - Derive local gradients of each node's output w.r.t. its immediate inputs (simple derivations)
- Forward propagation
    - Values are supplied to input variables
    - For each node that has values for all of its inputs, compute output and propagate forward
    - Repeat until all node outputs computed
- Backward propagation
    - Compute input gradient on the output nodes
    - For each node that has a value for its output gradient, compute each input gradient using chain rule and propagate backwards
    - Repeat until all gradients computed
- From each node's perspective
    - 

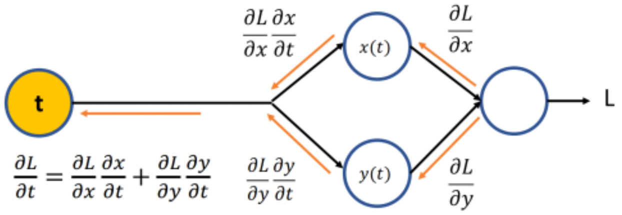
    - Forward propagation, when all input values arrive
        - Compute output value
        - Compute local gradient values
    - Backward propagation, when upstream gradient arrives on output
        - 

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial x_i} \qquad \frac{\partial L}{\partial f}$$

        - Using chain rule, compute downstream gradient on inputs
- Back propagation is a local process

- Computations for both forward and backward propagation can be performed on per-node basis as values arrive
    - On input during forward
    - On output during backward
- Local gradients can be computed during forward propagation
- Use chain rule to flow back

Gradients on different nodes

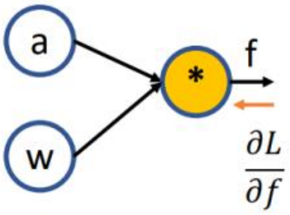
- Addition $f(x,y) = x + y$:
    - $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} = \frac{\partial L}{\partial f}$
    - Upstream gradient is distributed to all inputs
    - A change on any input independently changes the output
- Subtraction $f(x,y) = x - y$:
    - $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f}, \frac{\partial L}{\partial y} = -\frac{\partial L}{\partial f}$
    - Upstream gradient passed onto variables being subtracted from
    - Negative of upstream gradient passed onto variable being subtracted
- Multiplication $f(x,y) = xy$:
    - $\frac{\partial L}{\partial x} = y\frac{\partial L}{\partial f}, \frac{\partial L}{\partial y} = x\frac{\partial L}{\partial f}$
    - Upstream multiplied with all other input values
    - A change on an input is scaled by the value of the other inputs to affect a change in the output
- Equality (linear):
    - Pass through
- Branch:
    - 
$\frac{\partial L}{\partial x}\frac{\partial x}{\partial t}$
$x(t)$
$\frac{\partial L}{\partial x}$
t
L
$\frac{\partial L}{\partial t} = \frac{\partial L}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial t}$
$\frac{\partial L}{\partial y}\frac{\partial y}{\partial t}$
$y(t)$
$\frac{\partial L}{\partial y}$
    - Use the multivariable chain rule
- Max $f(x,y) = \max\{x,y\}$:
    - If $x > y, \frac{\partial L}{\partial x} = \frac{\partial L}{\partial f}, \frac{\partial L}{\partial y} = 0$
    - Upstream gradient is routed to larger variable
    - Only one input can affect the output at any time
- Sigmoid (softmax):
    - $\frac{\partial f}{\partial x} = f(1 - f).$
- Tanh:
    - $\frac{\partial f}{\partial x} = 1 - f^2.$

Back propagation at input layer

- No need to compute this, since we aren't interested in how to change the input to minimize loss
- But this can help visualize what the network has learned

Summary:

- Once upstream gradient is 0, all downstream gradients are also 0
- Back propagation sends a signal back throughout the network telling us how to change each parameter, but it doesn't make any neural network trainable



$$\frac{\partial f}{\partial w} = a, \qquad \frac{\partial f}{\partial a} = w$$

$$\frac{\partial L}{\partial w} = \frac{\partial f}{\partial w}\frac{\partial L}{\partial f} = a\frac{\partial L}{\partial f}$$

Activation functions

- Sigmoid:
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Maps input to values between 0 and 1
  - Vanishing (saturated) gradients (big problem)
    - When $|x|$ is large, the gradient is practically 0, which makes $\frac{\partial L}{\partial x} \to 0$ (saturated)
      - When in saturated region, it is a saturated neuron
      - Active (unsaturated) region is small
    - When gradient is small, learning will be slow
      - Parameters will change extremely slowly
      - Once a sigmoid neuron is in saturation, very hard for training to update the neuron's weights to improve the model
  - Always positive
    - All $\frac{\partial L}{\partial w_{i,j}^{[l]}}$ will be positive (have the same sign).
    - If all inputs to a unit are the same sign, then all weights for that unit have the same sign for $\frac{\partial L}{\partial w}$ (positive due to Sigmoid)
      - Gradient descent will update all weights in the same direction (all increase, all decrease)
    - Problem of Non-zero-centered inputs (inconvenient)
  - Max value of sigmoid gradient = 0.25
    - Each time gradients flow through a sigmoid function, it is reduced to $\frac{1}{4}$ or more
    - Also contributes to the vanishing gradients problems
  - Do not use Sigmoid for hidden layers
    - Can still use it on the output. With binary cross-entropy loss, the saturation effect is removed
    - Sigmoid function is a class of functions with the S shape
- Tanh activation function
  - $\tanh x = 2\sigma(2x) - 1$
  - Also a type of Sigmoid function
    - Still has saturated regions and vanishing gradients problem
  - Output range: $[-1,1]$
    - Solves the problem of non-zero-centered outputs
  - Generally faster learning compared to logistic sigmoid.
- Rectified Linear Activation Unit (ReLU)
  - $f(x) = \max(x, 0)$
  - Local gradient: $\frac{\partial f}{\partial x} = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$.
    - Downstream gradient $\frac{\partial L}{\partial x} = \begin{cases} \frac{\partial L}{\partial f}, x \geq 0\ (pass\ through) \\ 0, x < 0\ (no\ gradient) \end{cases}$.

  - Pros:
    - No vanishing gradient problem
    - Passthrough for gradient flow
    - Easy to compute
      - Speeds up training
      - Speeds up prediction
    - Sparse activations
      - ReLU can output a true 0
        - Sigmoid can only output near 0
        - Tanh can only output zero at one specific point
      - True 0 lead to sparse activations of neurons
  - Cons
    - Dead ReLU
      - If no gradient flows through a ReLU neuron, its associated parameters won't receive info on how to change
      - If this is the case for all training samples, then the parameters will never update
      - Cause $a_i^{[l]} = relu\left(z_i^{[l]}\right), z_i^{[l]} = W_i^{[l]} a^{[l-1]} + b_i^{[l]}$:
        - $z_i^{[l]} < 0$ for all training samples.
        - When $W_i^{[l]}, b_i^{[l]}$ initialized such that $z_i^{[l]} < 0$, dead from start.
        - Learning rate is too high. During iteration, $W_i^{[l]}, b_i^{[l]}$ updated such that $z_i^{[l]} < 0$.
      - Avoiding Dead ReLU
        - Initialize bias terms with small positive value
        - Need to be mindful about how we initialize weight parameters
    - Non-zero-centered output (all positive)
      - Not a big issue
  - When in doubt, use ReLU for FC NNs and CNNs
  - Need to be careful for RNNs due to exploding gradient problem
  - Variations
    - Try to fix dead ReLU problem by changing the $x < 0$ region
    - Leaky ReLU: $f(x) = \max(x, 0.01x)$
      - Gives a chance to get out of dead ReLU
    - Parametric ReLU (generalization of leaky ReLU)
      - $f(x) = \max(x, ax)$
      - Slope of line at $x < 0$ is a learned parameter
    - ELU: $f(x) = \begin{cases} x, x \leq 0 \\ a(e^x - 1), x > 0 \end{cases}$
    - SELU: $f(x) = \begin{cases} \lambda x, x \leq 0 \\ \lambda a(e^x - 1), x > 0 \end{cases}$
- Summary
  - ReLU is a good default choice
  - ReLU is strictly better than tanh
  - ReLU and tanh are strictly better than Sigmoid (Don't use Sigmoid for hidden layer)

Vectorized forward propagation

- Weight matrix: $\left(n^{[l]}, n^{[l-1]}\right)$.
- Bias vector: $\left(n^{[l]}, \right)$.
- Output vector: $\left(n^{[l]}, \right)$.
- $Z = WX + B$: $\left(n^{[l]}, \right)$.
- Activation: $A = g(Z)$.

Vectorized backward propagation

- Jacobian matrix $\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_m}{\partial x_2} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_1}{\partial x_n} & \frac{\partial f_2}{\partial x_n} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$
- Cost function: $\frac{\partial J}{\partial f} = \begin{pmatrix} \frac{\partial J}{\partial f_1} \\ \frac{\partial J}{\partial f_2} \\ \dots \\ \frac{\partial J}{\partial f_{n_f}} \end{pmatrix}$ shape $\left(n_f, 1\right)$,
  - Cross-entropy loss gives: $\frac{dJ}{d\hat{y}} = -\frac{1}{m}\frac{y}{\hat{y}}$.
- $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial x}$ shpe $\left(n_x, 1\right)$.
- Activation function shape: $(n,)$.
  - $\frac{\partial A}{\partial Z} = \begin{pmatrix} \frac{\partial a_1}{\partial z_1} & \frac{\partial a_2}{\partial z_1} & \dots & \frac{\partial a_n}{\partial z_1} \\ \frac{\partial a_1}{\partial z_2} & \frac{\partial a_2}{\partial z_2} & \dots & \frac{\partial a_n}{\partial z_2} \\ \dots & \dots & \dots & \dots \\ \frac{\partial a_1}{\partial z_n} & \frac{\partial a_2}{\partial z_n} & \dots & \frac{\partial a_n}{\partial z_n} \end{pmatrix}$.
  - For tanh activation
    - Since $a_1 = \tanh z_1, \dots. a_{n_h} = \tanh z_{n_h}$
    - $\frac{\partial A}{\partial Z} = \begin{pmatrix} \frac{\partial a_1}{\partial z_1} & 0 & \dots & 0 \\ 0 & \frac{\partial a_2}{\partial z_2} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{\partial a_n}{\partial z_n} \end{pmatrix}$.
    - $\frac{\partial a_i}{\partial z_i} = 1 - a_i^2$.
  - For ReLU
    - $\frac{\partial J}{\partial z_i} = \begin{cases} \frac{\partial J}{\partial a_i}, z_i \geq 0 \\ 0, z_i < 0 \end{cases}$.
    - Simply copy over upstream gradient or set to 0
  - For Softmax
    - $\frac{\partial J}{\partial z_i} = a_i\left(\frac{\partial J}{\partial a_i} - \left(\frac{\partial J}{\partial a}\right)^T \cdot a\right)$.
- Jacobian is diagonal (hence sparse) for element-wise vector operations
- Most vector operations used in neural networks have sparse Jacobian matrices
- We do not need to construct the full Jacobian matrix and never have to compute its full matrix-vector multiply with the upstream gradients

Tensors

- Multidimensional arrays
  - Scalar is 0d tensor
  - Vector is 1d tensor
  - Matrix is 2d tensor
- Local derivatives are high-order tensors
  - $f: (n_f, m_f), x: (n_x, m_x), y: \left(n_y, m_y\right)$.
  - $\frac{\partial f}{\partial x}: \left(n_x, m_x, n_f, m_f\right), \frac{\partial f}{\partial y}: (n_y, m_y, n_f, m_f)$.
  - $\frac{\partial J}{\partial f}: (n_f, m_f), \frac{\partial J}{\partial x} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial x}$.
- Derivative of a matrix by a scalar

  - $\frac{\partial F}{\partial x} = \begin{pmatrix} \frac{\partial f_{1,1}}{\partial x} & \frac{\partial f_{1,2}}{\partial x} & \cdots & \frac{\partial f_{1,m}}{\partial x} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_{n,1}}{\partial x} & \frac{\partial f_{n,2}}{\partial x} & \cdots & \frac{\partial f_{n,m}}{\partial x} \end{pmatrix}.$
- Each element of downstream gradient is inner product between slice of Jacobian and upstream gradient. But only one non-zero row
- Furthermore, Jacobian slices are just copies of rows from x, so we just need x

Cost function back propagation

- Downstream gradients will be scaled by $\frac{1}{m}$
- $\frac{\partial J}{\partial L} = \begin{pmatrix} \frac{1}{m} \\ \frac{1}{m} \\ \cdots \\ \frac{1}{m} \end{pmatrix}.$
- Each sample is only making a $\frac{1}{m}$ contribution to the final cost

Broadcasting (addition of the bias)

- $\frac{\partial J}{\partial B^{[l]}} = \frac{\partial J}{\partial Z}.$
- But $B^{[l]}$ is shape $(n^{[l]},)$, and $\frac{\partial J}{\partial Z}$ is shape $(n^{[l]}, m)$.
- We broadcast/replicate the bias to match the shape of $\frac{\partial J}{\partial Z}$
  - The same parameters are used for each of the m samples
- Each column is for one sample, each row is for one unit of the layer
  - $\frac{\partial J}{\partial b_i^{[l]}} = \left(\frac{\partial J}{\partial z_i^{[l]}}\right)^{(1)} + \left(\frac{\partial J}{\partial z_i^{[l]}}\right)^{(2)} + \cdots + \left(\frac{\partial J}{\partial z_i^{[l]}}\right)^{(m)} \sum_{j=1}^{m} \left(\frac{\partial J}{\partial z_i^{[l]}}\right)^{(j)}.$
  - Average loss is more practical than total loss

CNN

October 9, 2021 10:10 PM

Convolutional Neural Networks (CNNs)

- A class of neural networks typically used for image analysis and computer vision
- Image classification
- Retrieval
- object detection
- object segmentation
- Scene labeling
- Pose estimation
- Vision based reinforcement learning
- Image captioning

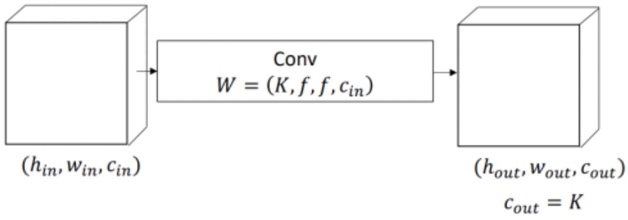
Image

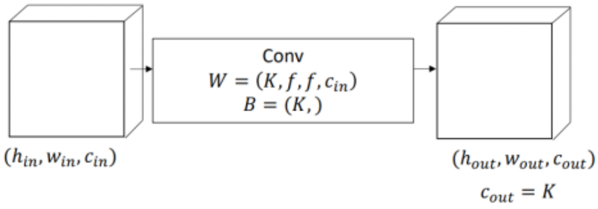
- data is unstructured data
- Converting to a feature vector throws away spatial information
- Too many parameters in fully connected network for large images
- Pixels that form a visual feature are local
    - Every unit is trying to make sense of the entire image
    - But spatial correlation is fairly local
    - Solution: locally connected
        - Have each unit connect only to a smaller region of the image
        - Can work well on centered images
        - No tolerance to translation
            - Also not taking advantage that image patterns often repeat at other parts of the image
    - Solution: shared patterns
        - Instead of multiple neurons sharing parameter, we use one neural that scans a specific feature (kernel)
        - Translation invariant
        - Use multiple filters. Each looks for a different feature

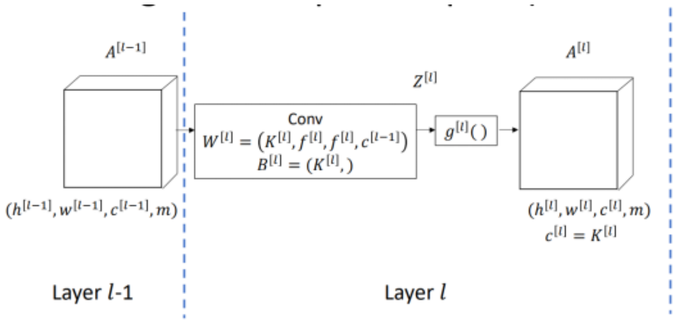
Convolutional filters

- Motivation: edge detection
- Operation: element-wise multiply and sum
- e.g. vertical edge detection
    - Kernel: $\begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$.
    - Output is intensity with which vertical edge occurs at the corresponding input location
    - If not high contrast, the intensity is lower
    - Dark to light: sign is different
        - Change signs on the filter
- Horizontal edge detection
    - Similar to vertical
    - Kernel: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$.
- Output:
    - Output of convolution is a feature map
    - Describes the intensity and location where a feature is present in the input image
- Treat the filter values as learnable parameters, supply data and let the model learn the best values for the data

Convolutional layer

- Each filter generates one feature map
    - Can think of each filter as a neuron
    - K number of (f, f, c), where f is the filter size and c is the number of color channel
- Collection of filters can be represented as a single (K, f, f, c) weight tensor
    - 

- Need a single bias for each filter with implicit broadcast
    - 

- Activation is applied to each element separately
- Total number of parameters in a convolutional layer
    - Weight parameters: $Kffc_{in}$.
    - Bias parameters: K.
    - Total: $K(ffc_{in} + 1)$.
- Generalization and Vectorization
    - 
$A^{[l-1]}$
$Z^{[l]}$
$A^{[l]}$
Conv
$W^{[l]} = (K^{[l]}, f^{[l]}, f^{[l]}, c^{[l-1]})$
$B^{[l]} = (K^{[l]},)$
$g^{[l]}()$
$(h^{[l-1]}, w^{[l-1]}, c^{[l-1]}, m)$
$(h^{[l]}, w^{[l]}, c^{[l]}, m)$
$c^{[l]} = K^{[l]}$
Layer l-1
Layer l
    - $h^{[l]} = h^{[l-1]} - f^{[l]} + 1.$
    - $w^{[l]} = w^{[l-1]} - f^{[l]} + 1.$
    - $c^{[l]} = K^{[l]}.$
- Filters look across all channels
    - Each channel of a volume is the activation map of a lower level feature
    - To build filters that look for compositions of lower level features, must look at multiple activation maps
- Filter the same shape as the input
    - Result will be a single number
    - Each filter corresponds to a single FC neuron

CNN and FC

- CNN is more efficient than FC
- CNN allows us to achieve sparse connectivity between layers while also taking advantage of spatial structure of image data to allow parameter sharing
    - Sparsely connected: each neuron is connected to a different subset of the inputs
    - Parameter sharing: instead of each neuron having its own weight and bias, they share the same parameters
- CNN is just a FC layer with sparse connectivity and parameter sharing
    - $a^{[l]} = g(conv(W^{[l]}, a^{[l-1]}) + b^{[l]}).$

Padding

- CNN shrinks the images in spatial dimensions of h, w.
  - Shrinking volumes
  - Input data at the edges influence fewer output values than input data in the middle
- Pad the perimeter of the input volume before convolution
  - Output preserves original spatial dimensions
  - Output dimension: $(h + 2p - f + 1, w + 2p - f + 1)$.
- Typically
  - No padding
  - Pad so that the output volume is the same as the input volume
    - $p = \frac{f-1}{2}$ only depends on the filter size
    - Works well for odd sizes, but causes asymmetry for even sizes. (Use only filters with odd size)

Stride

- Slide the convolutional filter by larger steps
- The amount by which we step is stride (s)
- Output size
  - Input: (h, w, c)
  - Output: $\left(\frac{h+2p-f}{s} + 1, \frac{w+2p-f}{s} + 1, K\right)$.
- A form of compression/down sampling of the feature map
- A way to shrink the volumes in a controlled fashion
  - It is necessary to control size before the final layer

Summary of convolutional layer

- Hyperparameters
  - Number of filters K
  - Filter size (f, f)
  - Stride s
  - Padding p
- Input volume
  - $\left(h^{[l-1]}, w^{[l-1]}, c^{[l-1]}\right)$
- Output volume
  - $\left(\frac{h^{[l-1]}+2p-f}{s} + 1, \frac{w^{[l-1]}+2p-f}{s} + 1, K\right)$.
- # learned parameters
  - $K(ffc^{[l-1]} + 1)$

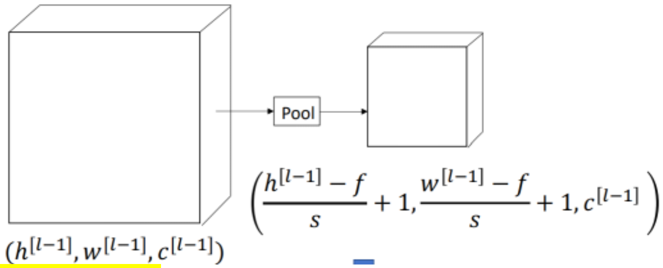
Receptive fields

- Suppose we use 3×3 filters in all layers
- Each output element sees a 3×3 region of its input
  - $1 \to 3 \times 3 \to 5 \times 5 \to 7 \times 7 \to \cdots$.

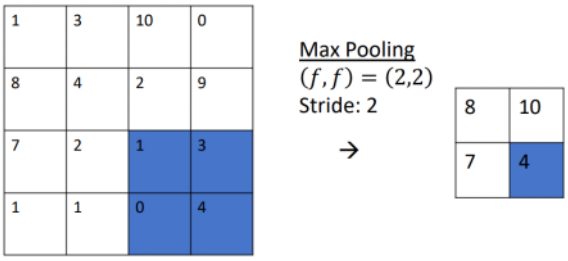
Final layers

- Image classifier
  - Flatten the final volume
  - Use one or more fully connected layer
  - Final volume must be a manageable size
  - Can think of convolutional layers as a feature extractors
    - Compress the image into a signature
    - Use the signature for classification
    - Learn structure from unstructured data
- Control size
  - Stride, padding

Pooling layer

- Pool each channel independently
  - Does not change channel size
  - Only changes spatial dimensions
- Hyperparameters
  - Pooling function
  - Pool size (f, f)
  - Stride s
  - No learned parameters
    - Reduces spatial dimensions, but does not change channel dimension
  - 

    $(h^{[l-1]}, w^{[l-1]}, c^{[l-1]})$

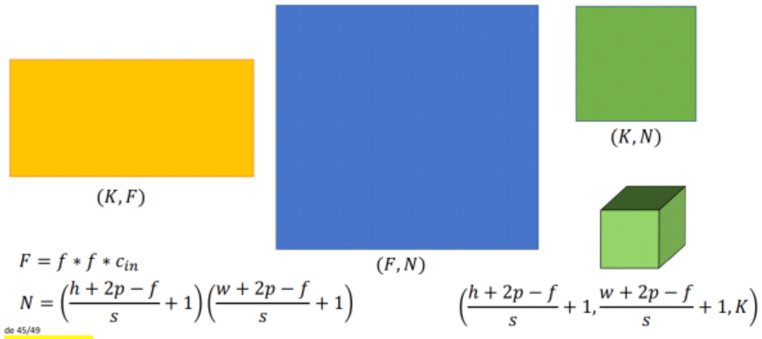
    $\left(\frac{h^{[l-1]} - f}{s} + 1, \frac{w^{[l-1]} - f}{s} + 1, c^{[l-1]}\right)$
  - Usually $f = s$, we have $\left(\frac{h^{[l-1]}}{s}, \frac{w^{[l-1]}}{s}, c^{[l-1]}\right)$
- Max pooling (used more)
  - Output is max value within each region
  - 

1	3	10	0
8	4	2	9
7	2	1	3
1	1	0	4

    Max Pooling
    $(f, f) = (2,2)$
    Stride: 2
    →

8	10
7	4
  - Reduces size (compress the data)
  - Discard all but the strongest signal
  - Adds flexibility to feature detection in the form of tolerance to translation
- Average pooling
  - Output is average value within each region

Vectorized Implementation

- Convolutions are implemented as matrix multiplication
- Transform input volume into 2D matrix
  - This depends on filter shape
  - Each "filter shape" elements forms a column in the matrix
- Transform filters into 2D matrix
  - Reshape the (f, f, c) filters into a row vector of size $(1, ffK)$
  - If there are K filters, each filter is a row in the matrix
- 

  $F = f * f * c_{in}$

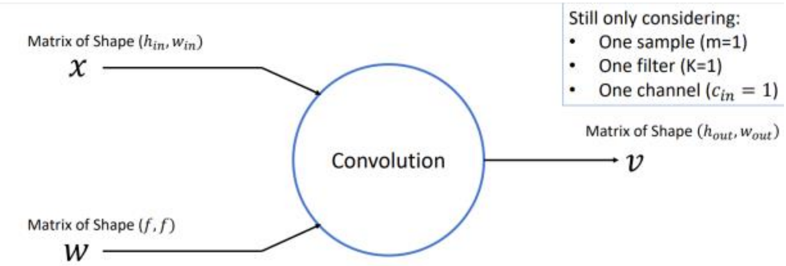
  $N = \left(\frac{h + 2p - f}{s} + 1\right)\left(\frac{w + 2p - f}{s} + 1\right)$

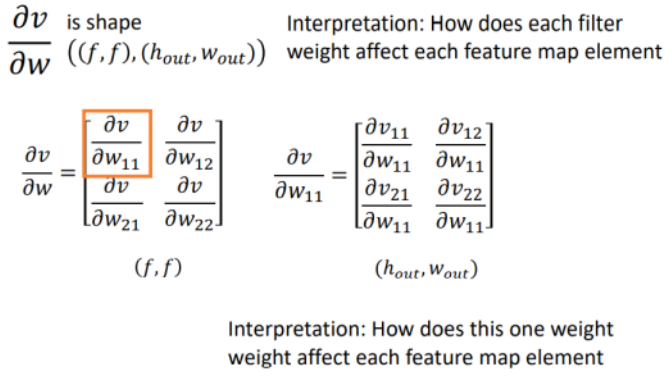
  $\left(\frac{h + 2p - f}{s} + 1, \frac{w + 2p - f}{s} + 1, K\right)$
- In code

- Transforming the input volume: im2col.
  - hard
- Transforming the weight matrix: w.reshape(K,-1).
- Transforming the final output is also a reshape

- Fourier transform
  - Convolution Theorem: $F\{f * g\} = F\{f\} \cdot F\{g\}$
  - Fourier transform of a convolution of two signals is equal to the elementwise product of the Fourier transform of each respective signal
    - $V_{in} * w = F^{-1}\left\{F\{V_{in}\} \cdot F\{w\}\right\}$.

Back propagation

- Convolution node
  - 
    Matrix of Shape (h_{in}, w_{in})
x
Matrix of Shape (f, f)
w
Convolution
Matrix of Shape (h_{out}, w_{out})
v

    Still only considering:
    - One sample (m=1)
    - One filter (K=1)
    - One channel ($c_{in} = 1$)
  - $\frac{\partial v}{\partial x}$ shape: $(h_{in}, w_{in}, h_{out}, w_{out})$.
    - $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial x} = pad\left(\frac{\partial J}{\partial v}, 0\right) * w.rotate(180)$.
    - Pad:
      - □ $h'_{out} = h_{out} + 2p_h, p_h = f - 1$.
      - □ $w'_{out} = w_{out} + 2p_w, p_w = f - 1$.
  - $\frac{\partial v}{\partial w}$ shape: (f, f, h_{out}, w_{out}).
    - $\frac{\partial J}{\partial w} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial w}$.
    - Compute each channel independently
    - 

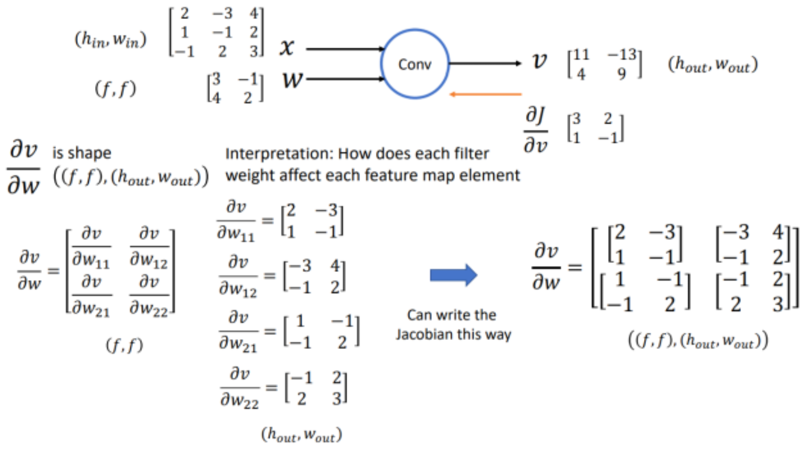
      $\frac{\partial v}{\partial w}$ is shape $((f, f), (h_{out}, w_{out}))$

      Interpretation: How does each filter weight affect each feature map element

      $$\frac{\partial v}{\partial w} = \begin{bmatrix} \boxed{\frac{\partial v}{\partial w_{11}}} & \frac{\partial v}{\partial w_{12}} \\ \frac{\partial v}{\partial w_{21}} & \frac{\partial v}{\partial w_{22}} \end{bmatrix} \qquad \frac{\partial v}{\partial w_{11}} = \begin{bmatrix} \frac{\partial v_{11}}{\partial w_{11}} & \frac{\partial v_{12}}{\partial w_{11}} \\ \frac{\partial v_{21}}{\partial w_{11}} & \frac{\partial v_{22}}{\partial w_{11}} \end{bmatrix}$$

      (f, f) (h_{out}, w_{out})

      Interpretation: How does this one weight weight affect each feature map element
    - Here, $\frac{\partial v}{\partial w_{11}} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$.


(h_{in}, w_{in}) $\begin{bmatrix} 2 & -3 & 4 \\ 1 & -1 & 2 \\ -1 & 2 & 3 \end{bmatrix}$ x
(f, f) $\begin{bmatrix} 3 & -1 \\ 4 & 2 \end{bmatrix}$ w
Conv
v $\begin{bmatrix} 11 & -13 \\ 4 & 9 \end{bmatrix}$ (h_{out}, w_{out})
$\frac{\partial J}{\partial v}$ $\begin{bmatrix} 3 & 2 \\ 1 & -1 \end{bmatrix}$

- $\frac{\partial v}{\partial w}$ is shape $((f, f), (h_{out}, w_{out}))$

  Interpretation: How does each filter weight affect each feature map element

  $$\frac{\partial v}{\partial w} = \begin{bmatrix} \frac{\partial v}{\partial w_{11}} & \frac{\partial v}{\partial w_{12}} \\ \frac{\partial v}{\partial w_{21}} & \frac{\partial v}{\partial w_{22}} \end{bmatrix}$$

  (f, f)

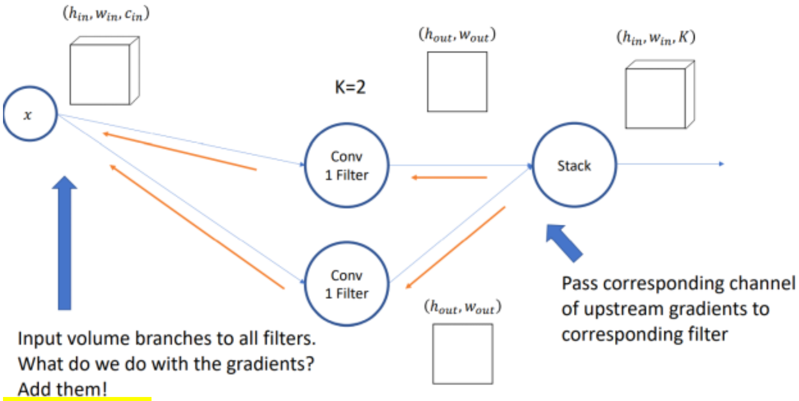
  $$\frac{\partial v}{\partial w_{11}} = \begin{bmatrix} 2 & -3 \\ 1 & -1 \end{bmatrix} \quad \frac{\partial v}{\partial w_{12}} = \begin{bmatrix} -3 & 4 \\ -1 & 2 \end{bmatrix} \quad \frac{\partial v}{\partial w_{21}} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \quad \frac{\partial v}{\partial w_{22}} = \begin{bmatrix} -1 & 2 \\ 2 & 3 \end{bmatrix}$$

  (h_{out}, w_{out})

  Can write the Jacobian this way

  $$\frac{\partial v}{\partial w} = \begin{bmatrix} \begin{bmatrix} 2 & -3 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} -3 & 4 \\ -1 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} & \begin{bmatrix} -1 & 2 \\ 2 & 3 \end{bmatrix} \end{bmatrix}$$

  $((f, f), (h_{out}, w_{out}))$

        - $\frac{\partial J}{\partial w} = \frac{\partial v}{\partial w}\frac{\partial J}{\partial v} = \begin{bmatrix} \begin{bmatrix} 2 & -3 \\ 1 & -1 \end{bmatrix} & \begin{bmatrix} -3 & 4 \\ -1 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} & \begin{bmatrix} -1 & 2 \\ 2 & 3 \end{bmatrix} \end{bmatrix} \begin{bmatrix} 3 & -2 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 12 & -20 \\ 2 & -8 \end{bmatrix}$
            - □ Each Jacobian slice is a sliding window over the input x
            - □ $\frac{\partial J}{\partial w} = x * \frac{\partial J}{\partial v}$.
        - For FC layer, Jacobian has a lot of 0
            - □ Each neuron has own set of weights.
            - □ They do not affect the output of other neurons
        - For Conv layer, every weight affects every output
- Chain rule application: tensor-matrix multiply
- 

  Input volume branches to all filters. What do we do with the gradients? Add them!

  Pass corresponding channel of upstream gradients to corresponding filter

- Max pooling
    - ○ Upstream gradient is routed to larger variable
    - ○ Only one input can affect the output at any time
    - ○ Similar to max function

Adversarial inputs via back propagation

- Pick an input image to modify
- Pick an output class you want to trick the classifier into predicting
- Use a cost function that maximizes that class's output probability
- Use back propagation to find changes to the input image to maximize cost

Numerical gradient checking

- $\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$.
- When h is not zero, but very small, we can get a decent approximation to the derivative
- For a multivariable function
    - ○ $\frac{\partial y}{\partial x_1} = \frac{f(x_1+h, x_2, \ldots, x_n) - f(x_1-h, x_2, \ldots, x_n)}{2h}$.
    - ○ $\frac{\partial y}{\partial x_n} = \frac{f(x_1, x_2, \ldots, x_n+h) - f(x_1, x_2, \ldots, x_n-h)}{2h}$.

Number of parameters per layers

- Convolutional layer: $K(f * f * c_{in} + 1)$
- Max pooling: 0
- Fully connected: $n_h^{[l]}\left(n_h^{[l-1]} + 1\right)$.

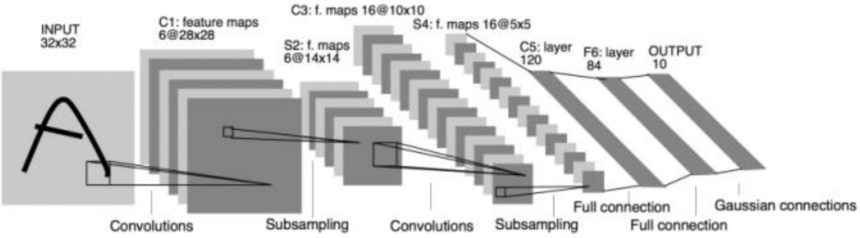
CNN architectures & applications

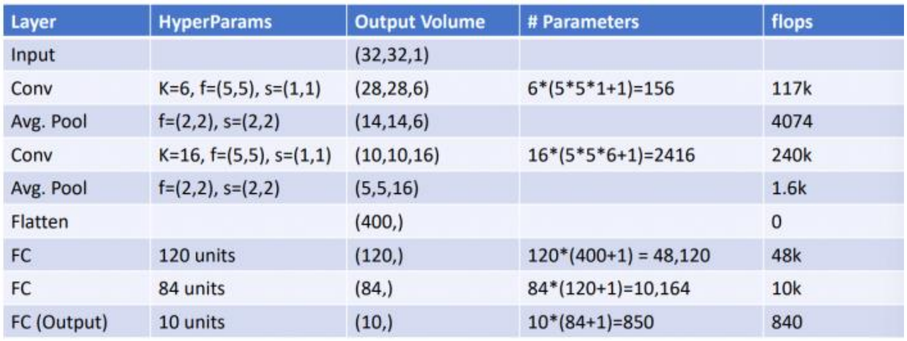
2021年11月4日 12:18

Computational resource analysis

- Number of floating point operations (FLOPs) for a convolution layer
    - Convolution is a bunch of multiply-accumulate (MAC) operations. One MAC can be done in a single flop
    - Given weights (K, f, f, c_{in}) and output of shape of $(h_{out}, w_{out}, c_{out})$
        - (h_{out}, w_{out}, K) activations to compute
        - Each activation is a dot product between two (f, f, c_{in}) tensors (MACs)
        - Total flops: $h_{out} * w_{out} * K * f * f * c_{in}$
            - Number of outputs * number of flops to compute each output
- Number of FLOPs for pooling layer
    - Given a single region (f, f) in which to pool
    - Max pool is comparison of $f * f$ numbers
    - Avg pool is addition of $f * f$ numbers
    - Total flops: $f * f$.
    - Given a pooling layer with output shape $(h_{out}, w_{out}, c_{out})$.
        - $h_{out} * w_{out} * c_{out}$ regions to compute.
        - Total flops: $h_{out} * w_{out} * c_{out} * f * f$.
- Number of FLOPs for FC layer
    - Output of each unit is weighted sum of $n_h^{[l-1]}$ numbers (MACs)
    - Output of all units (total flop) is $n_h^{[l]} * n_h^{[l-1]}$.
- FLOPs depends on a lot of implementation details
    - Hardware architecture
    - The way you write the code
    - Compiler

LeNet:





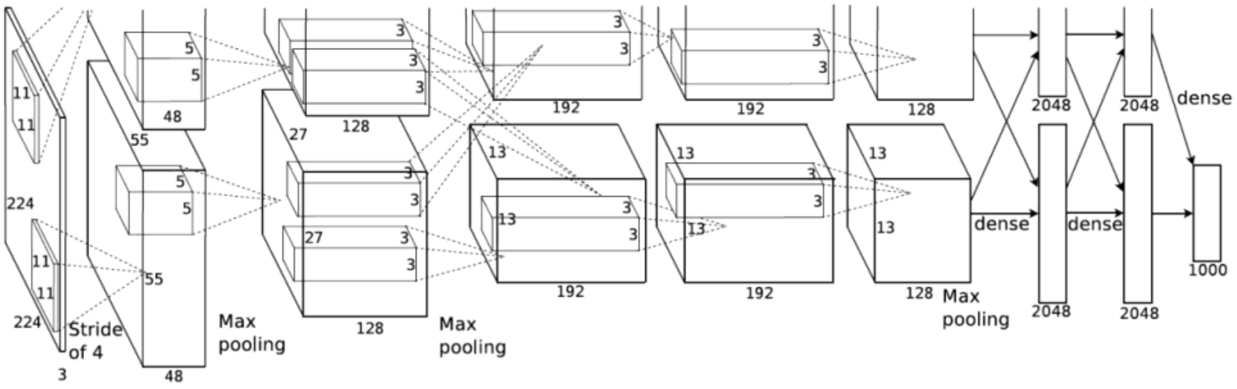
Layer	HyperParams	Output Volume	# Parameters	flops
Input		(32,32,1)		
Conv	K=6, f=(5,5), s=(1,1)	(28,28,6)	6*(5*5*1+1)=156	117k
Avg. Pool	f=(2,2), s=(2,2)	(14,14,6)		4074
Conv	K=16, f=(5,5), s=(1,1)	(10,10,16)	16*(5*5*6+1)=2416	240k
Avg. Pool	f=(2,2), s=(2,2)	(5,5,16)		1.6k
Flatten		(400,)		0
FC	120 units	(120,)	120*(400+1) = 48,120	48k
FC	84 units	(84,)	84*(120+1)=10,164	10k
FC (Output)	10 units	(10,)	10*(84+1)=850	840

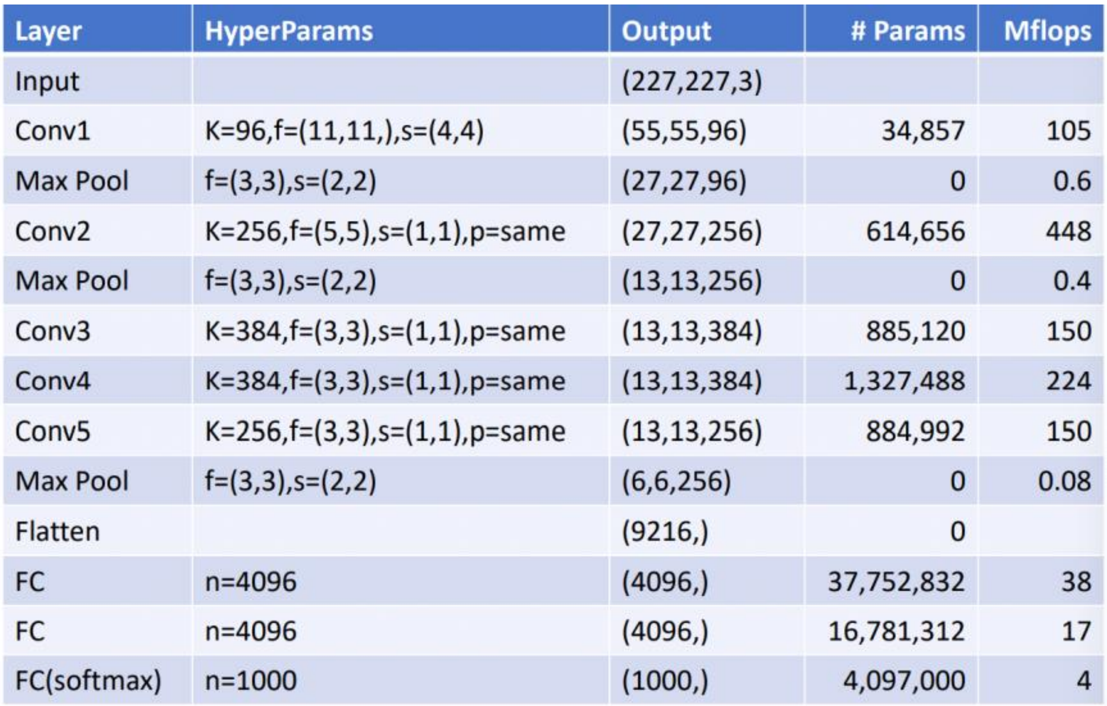
Top1 and Top5 error

- Top1: the fraction of test images for which the correct label is not the prediction of the model
- Top5: the fraction of test images for which the correct label is not among the five labels considered most probable by the model

Alex net

- Popularized CNNs for computer vision
- 16% top-5 error, 26% for runner up
- Popularized ReLUs for CNNs
  - Networks with ReLU consistently learned faster
- Overlapping pooling
  - Reduce top1 and top5 error
  - Overlapping pooling helped model generalize (reduce overfit)
- Used local response normalization layers
- Architecture hyperparameters chosen by trial-and-error

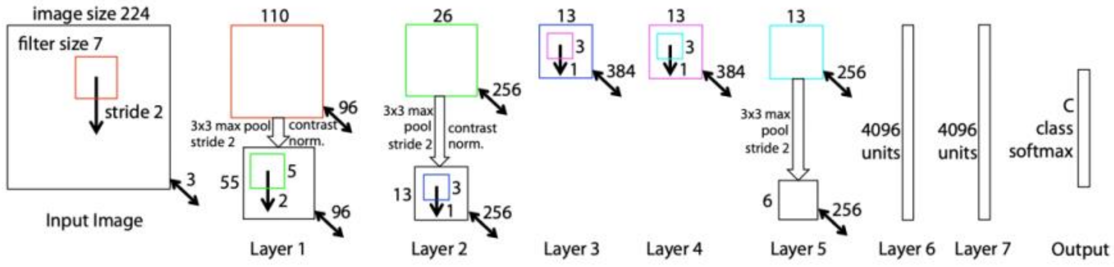




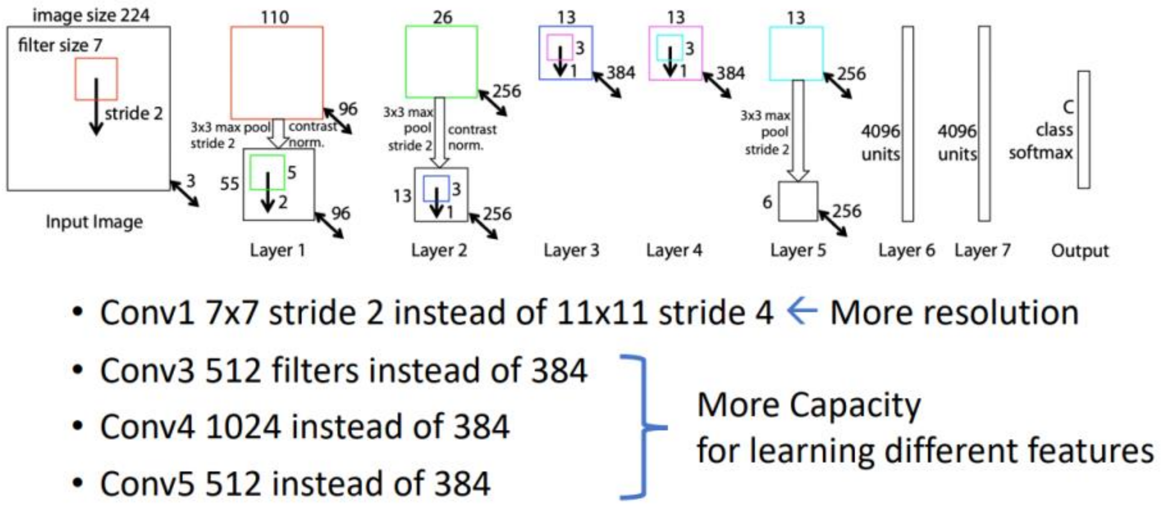
Layer	HyperParams	Output	# Params	Mflops
Input		(227,227,3)		
Conv1	K=96,f=(11,11,),s=(4,4)	(55,55,96)	34,857	105
Max Pool	f=(3,3),s=(2,2)	(27,27,96)	0	0.6
Conv2	K=256,f=(5,5),s=(1,1),p=same	(27,27,256)	614,656	448
Max Pool	f=(3,3),s=(2,2)	(13,13,256)	0	0.4
Conv3	K=384,f=(3,3),s=(1,1),p=same	(13,13,384)	885,120	150
Conv4	K=384,f=(3,3),s=(1,1),p=same	(13,13,384)	1,327,488	224
Conv5	K=256,f=(3,3),s=(1,1),p=same	(13,13,256)	884,992	150
Max Pool	f=(3,3),s=(2,2)	(6,6,256)	0	0.08
Flatten		(9216,)	0	
FC	n=4096	(4096,)	37,752,832	38
FC	n=4096	(4096,)	16,781,312	17
FC(softmax)	n=1000	(1000,)	4,097,000	4

ZFNet

- A bigger Alex net
- Bigger capacity is still better
- Still use trial-and-error for architecture design
- No consideration for computation efficiency

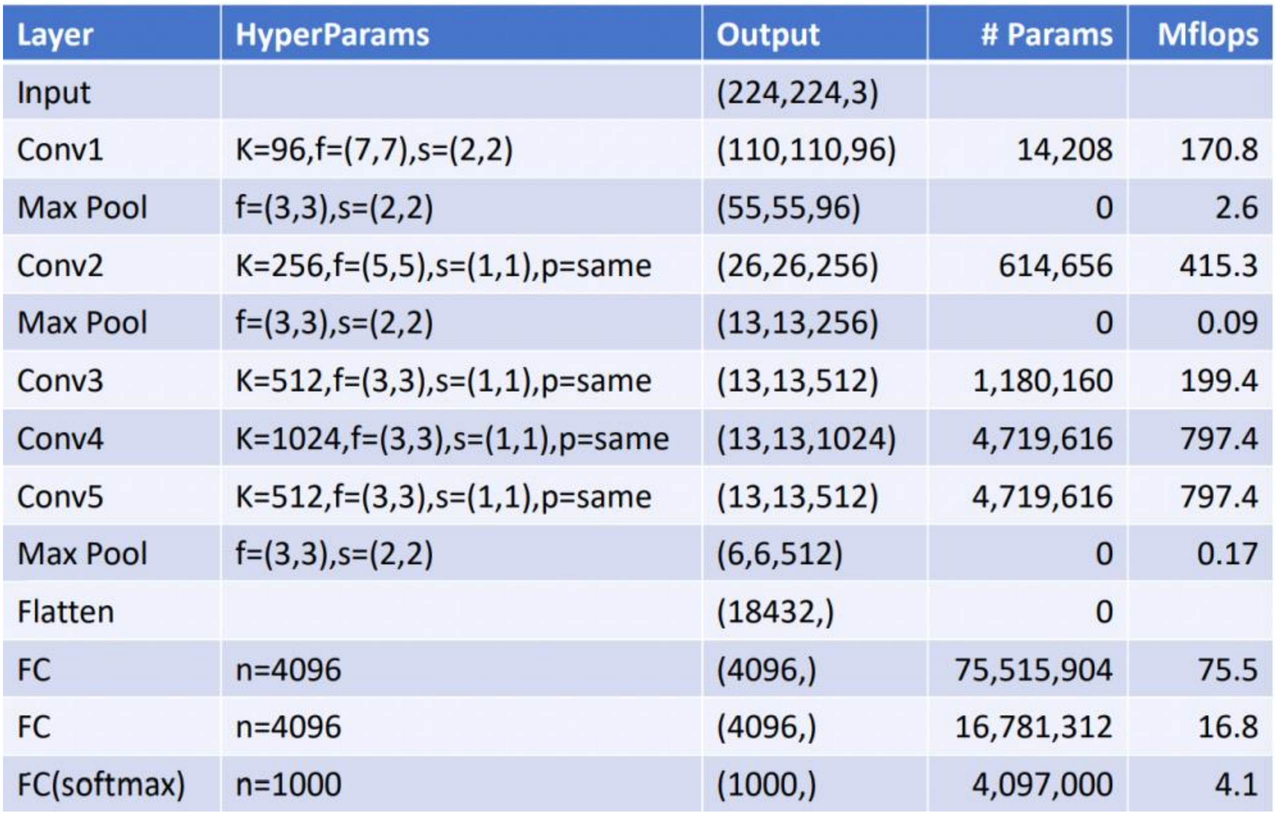


- Conv1 7x7 stride 2 instead of 11x11 stride 4 ← More resolution



- Conv1 7x7 stride 2 instead of 11x11 stride 4 ← More resolution
- Conv3 512 filters instead of 384
- Conv4 1024 instead of 384
- Conv5 512 instead of 384

More Capacity
for learning different features



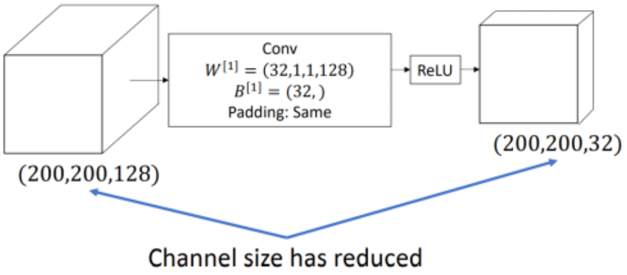
Layer	HyperParams	Output	# Params	Mflops
Input		(224,224,3)		
Conv1	K=96,f=(7,7),s=(2,2)	(110,110,96)	14,208	170.8
Max Pool	f=(3,3),s=(2,2)	(55,55,96)	0	2.6
Conv2	K=256,f=(5,5),s=(1,1),p=same	(26,26,256)	614,656	415.3
Max Pool	f=(3,3),s=(2,2)	(13,13,256)	0	0.09
Conv3	K=512,f=(3,3),s=(1,1),p=same	(13,13,512)	1,180,160	199.4
Conv4	K=1024,f=(3,3),s=(1,1),p=same	(13,13,1024)	4,719,616	797.4
Conv5	K=512,f=(3,3),s=(1,1),p=same	(13,13,512)	4,719,616	797.4
Max Pool	f=(3,3),s=(2,2)	(6,6,512)	0	0.17
Flatten		(18432,)	0	
FC	n=4096	(4096,)	75,515,904	75.5
FC	n=4096	(4096,)	16,781,312	16.8
FC(softmax)	n=1000	(1000,)	4,097,000	4.1

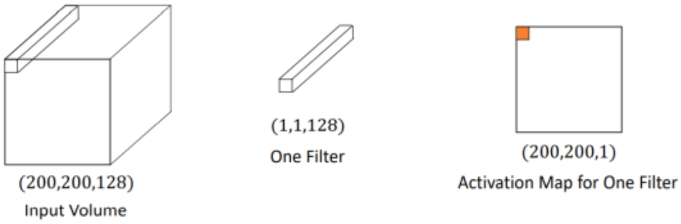
VGGNet

- Systematic design principles
    - All conv layers are 3 × 3 stride 1, same pad
        - Two stacked 3 × 3 conv layers can still see a 5 × 5 spatial region of the output
        - Two 3 × 3 layers use less parameters, less flops than one 5 × 5 layer, but needs more memory due to intermediate activation maps.
        - Still, stacking smaller filters is better
            - □ Can achieve equivalent receptive field
            - □ Fewer parameters to train
            - □ Requires less computation
            - □ Needs more memory, but not a problem with GPU memory
            - □ Has multiple levels of non-linearities (ReLU)
            - □ Less overfitting
    - All max pool layers are 2 × 2 stride 2
        - Necessary for controlling final volume size
        - Non-overlapping stride follows intuition of doing a straight-forward down-sampling

    - The conv layer following a pool layer will have enough filters to double the volume channel size
        - A conv layer operating on a volume that has half spatial dimensions and double channel size take the same number of flops
        - Keeps same compute time per layer
- VGGNet is a class of architectures
    - Using design rules, a number of architectures were evaluated
    - Each architecture has 5 stages
    - A stage consists of 1-4 conv layers followed by max pool
    - The ones that people talk about are VGG16 and VGG19, with 16 and 19 layers
- Summary
    - Very uniform and straight forward architecture
    - Has a large number of parameters
    - VGG19 slightly better than VGG16
    - Win the localization challenge, but not the classification challenge

GoogLeNet (Inception)

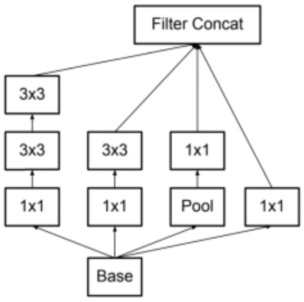
- Motivations
    - Efficient use of compute resources
    - Bigger architecture is potentially better, but
        - More parameters - more prone to overfitting - get more data - expensive
        - Requires more computation - computation budget is finite - need to be more efficient with how you go bigger
- Inception module
    - Basic building block of the inception network
    - VGGNet eliminated filter size as a hyperparameter by proposing to always use 3 × 3 and arguing that this has many benefits
    - Inception module eliminates filter size as a hyperparameter
    - Has filters of different sizes in a single layer
        - Stack the output into a single volume
    - Still computationally efficient
- 1 × 1 convolutions
    - Pooling allows us to down-sample/reduce the spatial dimensions, but doesn't let us change the size of the channel dimension
    - Can reduce the channel dimension using a convolution layer with 1 × 1 filters
    - May seem redundant, but filters have an implied third dimension equal to the input volumes number of channels
    - 

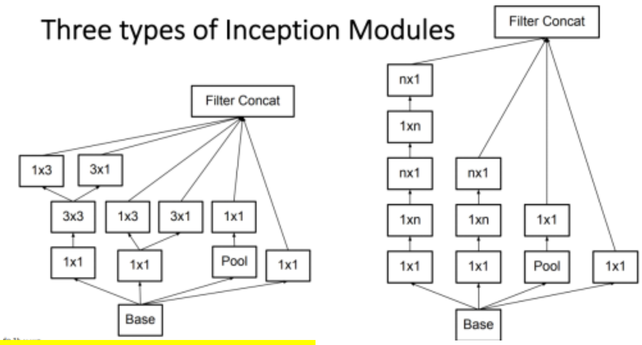
    - For one of the filters
        - Weighted sum across all feature maps at each spatial location
        - 

    - Conceptually like a form of compression where compression scheme is learned from the data

    - Output features are a composition of the input features
- Summary of inception module
    - Inception module has filters of different sizes in same layer
    - Use 1 × 1 convolutions to improve computation efficiency
    - Intuition of 1 × 1 convolutions is combining feature maps
    - Doesn't hurt as long as not too aggressive
- Global average pooling
    - Traditionally, final layers is a flattening of the final volume into a vector and sending this to one or more FC layers
        - Huge vector - large number of parameters for subsequent FC layer
    - Another approach
        - Average pool across the entirety of each activation map - one number per activation map
        - Resulting vector is fed to subsequent FC layers
    - Advantages
        - Pooling operation is essentially free
        - No parameters to optimize so less prone to overfitting
        - Since we are looking over the entire feature map, thus more robust to spatial translation of the final activations

InceptionV3(Reception)

- Three types of inception modules
    - First inception module
        - Same as GoogLeNet's inception module except 5 × 5 replaced by two layers of 3 × 3 filters
        - 

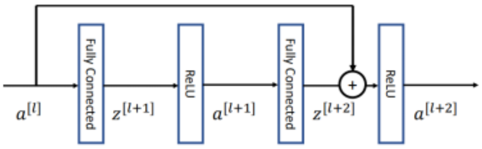
    - 

- Spatially separable convolutions
    - Decompose a 3 × 3 convolution into two convolutions (3 × 1 and 1 × 3)
    - More efficient than using one convolution.

ResNet

- Both training and test errors may increase with more layers
- Deep network should be at least as good as shallow network
    - If the additional layers just learned the identity, then functionally, the deeper network is equivalent to the shallow network
- Optimization problem
    - Hypothesis: current techniques make it hard to find the identity function for a layer and a function that improves the overall model
    - Proposed solution: augment architecture to start with the identity function, and then

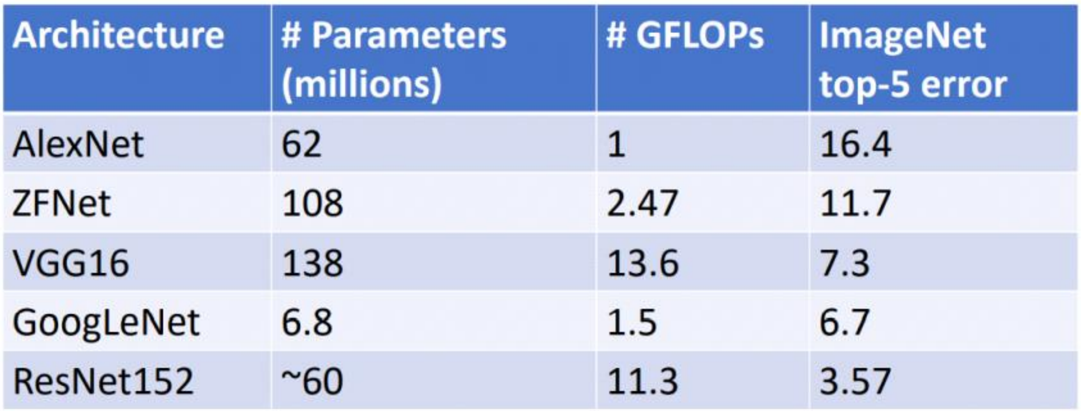
learn from there

- Residual block for fully connected layer
    - 

$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$
$$= ReLU\left(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}\right)$$

    - Add a shortcut
    - If $W^{[l+2]}$ and $b^{[l+2]}$ approach 0, then $a^{[l+2]} = a^{[l]}$.
    - Stacking these blocks to make a network deeper shouldn't hurt
    - The residual identity function gives a good baseline on which to try to improve
    - Also
        - Doesn't add any learned parameters
        - Doesn't increase computational complexity significantly
        - Shortcut paths provide another path for backprop gradient flow
    - Shape of $z^{[l+2]}$ and $a^{[l]}$ must match.
        - If not, either use a projection matrix or pad with zeros
- Architecture
    - 34 parameter layers
    - No pooling layers. Use stride=2 in conv layer to shrink volumes
    - Use global average pooling instead of FC layers at the end

Comparison



Architecture	# Parameters (millions)	# GFLOPs	ImageNet top-5 error
AlexNet	62	1	16.4
ZFNet	108	2.47	11.7
VGG16	138	13.6	7.3
GoogLeNet	6.8	1.5	6.7
ResNet152	~60	11.3	3.57

Memory usage

- Sources
    - Activations: the intermediate volumes and their gradients
    - Parameters: parameter values and their gradients
    - Training data: the batch currently being processed
- For training, you need to fit everything into the GPU memory, or else you take massive runtime hit
- Can tune optimizer batch size

MobileNet

- Another way of using 1×1 convolutions to create a factorized convolution which in turn further improves compute efficiency
- Hyperparameter to trade off accuracy and FLOPs/Params

Traditional convolution

- Filter produces a single map
    - Channel independent convolution
    - Summing across channels/1×1 convolution with fixed filter value (1).

- Depth-wise separable convolutions has two stages
    - Depth-wise convolution
        - One (f, f, c_{in}) filter
        - Each channel convolved independently
    - Point wise convolution
        - K number of $(1,1, c_{in})$ filters.

Object localization and detection

- Localization
    - Output
        - Class prediction
        - Bounding box b_x, b_y, b_w, b_h
        - Fixed number of objects
    - Start with CNN classifier architecture
    - Add FC layer to predict bounding box
        - Treat as regression problem
        - Use squared loss (i.e. L_2 loss)
            - □ $L\left(b_x, b_y, b_w, b_h, \widehat{b_x}. \widehat{b_y}, \widehat{b_w}, \widehat{b_h}\right) = \sum_{i \in \{x,y,w,h\}} \left(b_i - \widehat{b_i}\right)^2$.
        - Bounding box cost = average loss (with L_2 loss, mean squared error/MSE)
        - Final cost = categorical cross entropy loss (class prediction) + Bounding box cost
- Landmark detection
    - Localization with only the center x, y.
    - FC layer predicts two numbers (x, y) for each landmark.
    - Examples
        - Face detection
        - Pose detection: define a landmark for each joint
- Object detection
    - Detecting fixed number of objects: localization
    - Detecting multiple objects: sliding window
        - Start with a trained CNN classifier
        - Supply various crops of the image to the CNN via sliding window
        - Sliding window locations for one window of shape (b_h, b_w) in an image of shape (H, W):
            - □ $(H - b_h + 1) \cdot (W - b_w + 1)$.
        - Repeat for all possible window shapes:
            - □ $\sum_{b_h=1}^{H} \sum_{b_w=1}^{W} (H - b_h + 1) \cdot (W - b_w + 1)$.
            - □ Infeasible to look at all possible window sizes at all locations iteratively

Regions with CNN features (R-CNN)

- First use a region proposal algorithm to find a manageable number of regions (crops) that potentially have an object
- Send region crops to classifier
- Region crop location and size is the bounding box prediction
- R-CNN
    - Evaluate one region at a time
- Fast R-CNN
    - Classify all proposed regions at once
- Faster R-CNN
    - Uses a CNN to propose regions

You only look once (YOLO)

- Implement sliding window via convolution
    - Start with a trained CNN classifier
    - Convert FC layers to use convolutional equivalent implementation
    - Supply larger image for object detection

    - Each sliding window location is a potential bounding box for an object
        - For each output set, we can map back to region of input
- Can evaluate all sliding window locations in one pass
- Some restrictions on stride and size of the sliding window
- Conv layer to FC layer
    - Flatten
    - Convolve with filters that have the same shape as input volume, one filter for each FC output unit
- Anchor box
    - Change localizer to predict up to X objects at each location with predefined bounding box shapes

Problem of sliding window

- Objects may not fit perfectly inside of sliding window
    - Inaccurate bounding box predictions
- Solution
    - Instead of applying a CNN classifier at each sliding window location, apply a CNN classifier + localizer
        - Outputs a bounding box prediction in addition to class predictions
- Can only detect one object at each sliding window location

Image retrieval

- Use the final flattened volume as a signature of an image
- Find similar images by finding similar signatures
- With a trained network, compute and store signature vector of each image
- Given a new image, find images with the smallest Euclidian distance between signature vectors

Visualization feature vectors

- Flatten out last volume
- Apply dimension reduction
- plot

Saliency maps

- Define the parts of the image imported for the prediction
- Can do image segmentation

Gradient Descent & optimization

2021年11月16日 9:12

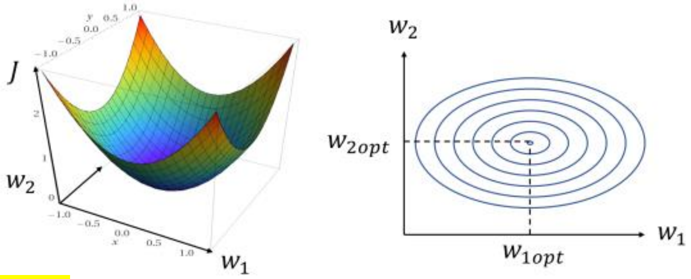
Optimizer:

- Get the network to reach its potential by finding good parameter values

Optimization

- Define a cost function (objective function) to measure the quality of a solution
  - Cost function models desired traits (objectives) of the solution
  - The solution is a set of parameter values
  - The objective is to minimize difference between model prediction and actual labels
- Use an optimization algorithm (optimizer) to search for a solution that minimizes/maximizes the cost function
  - Infeasible to solve for an optimal solution
  - Instead, iteratively search for a good-enough solution

Neural network classifier

- Objective is to minimize difference between true label y and predicted label $\hat{y}, J = f(y, \hat{y})$
- Prediction is a function of the input x and network parameters $w, b, \hat{y} = h(x, w, b)$.
- Training objective is thus a function of $y, x, w, b, J = f(y, x, w, b)$.
  - For a given training set, x, y are constant, $J = f(w, b)$.
- 

- Optima:
  - Minima:
    - Convex in all variables
  - Maxima:
    - Concave in all variables
  - Global refers to the biggest/smallest among all maxima/minima
  - Local refers to all the rest
  - Related but not optima (Saddle):
    - Concave in some variables, convex in others
  - Gradients at optima and saddle are 0, $\frac{dJ}{dw_i} = 0$ for all parameters w_i.

Deep learning cost function is not convex

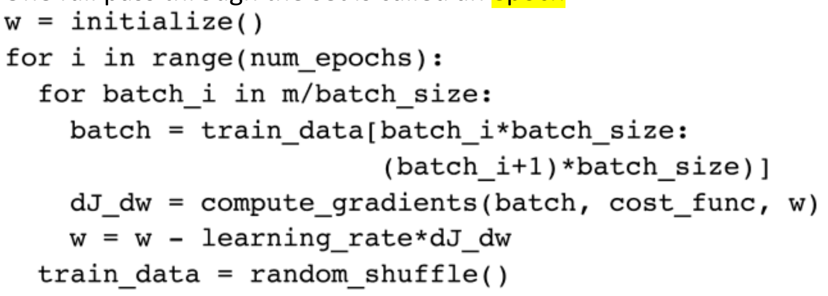
- There are many equal global minima

Gradient descent

- Intuition
  - Start somewhere in parameter space
  - Move in direction with the steepest decrease in cost
  - Repeat
- Hyperparameters
  - Parameter initialization method
  - Learning rate
  - Number of iterations
- Improving gradient descent allows us to go through training faster and tune more

- Problem
    - Cost is a function of all training image
    - When training set size gets large, computational requirements make classic gradient descent impractical
        - Takes too long to compute gradient for one training iteration
        - Requires too much memory to store activations of all samples concurrently in GPU memory

Mini-Batch gradient descent

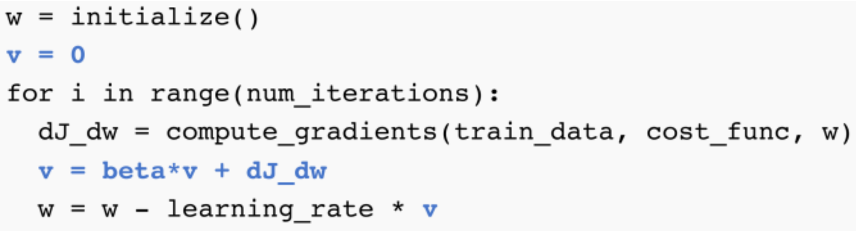
- Use a small subset of the training set (a mini-batch) as an approximation of the overall training set
- Hyperparameters
    - Parameter initialization method
    - Learning rate
    - Number of iterations
    - Sampling method
    - Batch size - 32/64/128/256
        - Power of 2 because sometimes memory access works out better
        - Pick as big as you can and still fit into GPU memory, significant performance hit from memory access if can't fit into memory
- A common sample method
    - Random shuffle full set
    - Partition into mini-batches
    - Iterate across each mini-batch
    - One full pass through the set is called an epoch
    - 

```
w = initialize()
for i in range(num_epochs):
  for batch_i in m/batch_size:
    batch = train_data[batch_i*batch_size:
                       (batch_i+1)*batch_size)]
    dJ_dw = compute_gradients(batch, cost_func, w)
    w = w - learning_rate*dJ_dw
  train_data = random_shuffle()
```

- If minibatch size=full size, same as classic gradient descent
- If minibatch size=1, each sample is a mini-batch
    - Stochastic gradient descent (SGD)
        - Keras use SGD to refer to mini-batch gradient descent
    - Lose benefits from vectorization
- Problems
    - Different dimensions (parameters) may change at different rates
        - Direction of steepest descent isn't directly to minimum unless it is a circle
        - Larger steps at steeper areas, and smaller steps at shallower areas
    - Local optima and saddle points
        - Saddle points are unstable but simply no gradient info
        - Gradient descent will stop updating parameters
    - Meandering nature of mini-batch gradient descent
        - Winding path

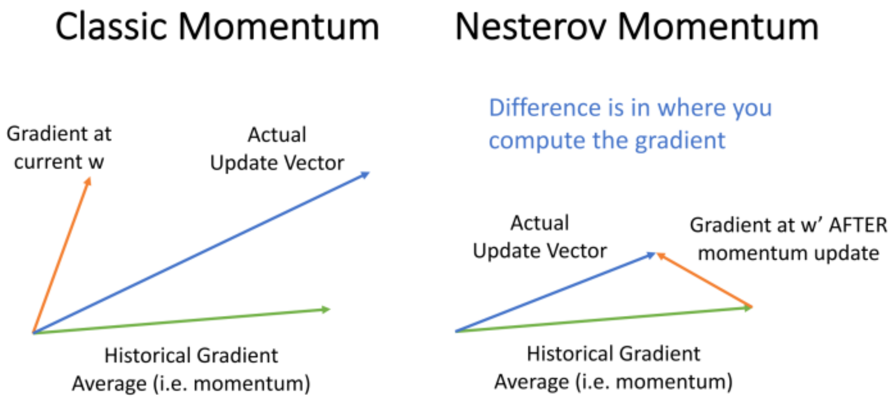
Exponentially weighted averages

- Moving average
- Can be used to smooth out short-term fluctuations and highlight longer-term trends
- $v_t = \beta v_{t-1} + (1-\beta)x_t.$
    - Approximately v_t is the average value over $\frac{1}{1-\beta}$ datapoints.
    - β slows down the descent
- Use an exponentially weighted average of past gradients to update the parameters

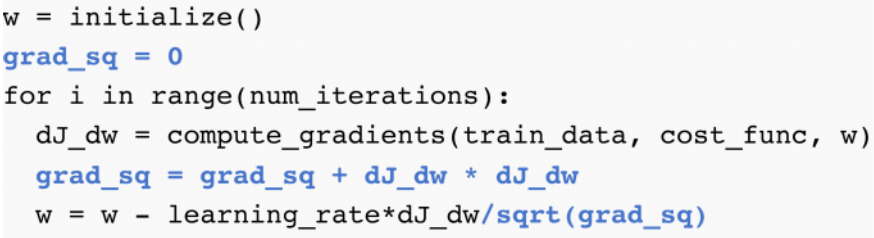
- 

  ```
  w = initialize()
  v = 0
  for i in range(num_iterations):
    dJ_dw = compute_gradients(train_data, cost_func, w)
    v = beta*v + dJ_dw
    w = w - learning_rate * v
  ```

    - $1-\beta$ doesn't matter too much (factored into learning rate)
    - $v_t = \beta v_{t-1} + \frac{\partial J}{\partial w}$.
- Solution to problem 1
  - Consistent gradient will build up velocity from accumulated acceleration
  - Inconsistent gradients will cancel out
- Solution to problem 2
  - At saddle points, gradient is 0, but historical component (momentum) won't be
  - At local minima, velocity can help get back out of some local minima
- Solution to problem 3
  - The moving average create a smoothing effect
- Bias correction (issue at t around zero)
  - $v_{t\_biased} = \beta \cdot v_{t-1_{biased}} + (1-\beta)T_t$.
  - $v_t = \frac{v_{t\_biased}}{1-\beta^t}$.
  - Biases will make large updates at the start which will destroy weight initialization or send you into a spot in the parameter space with no gradients

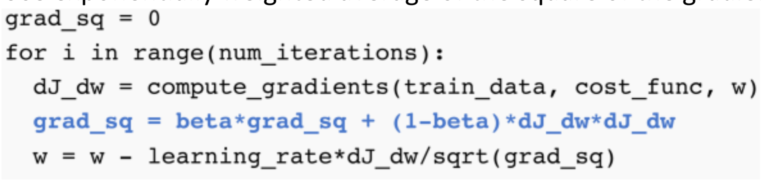

Classic Momentum
Nesterov Momentum
Gradient at current w
Actual Update Vector
Historical Gradient Average (i.e. momentum)
Difference is in where you compute the gradient
Actual Update Vector
Gradient at w' AFTER momentum update
Historical Gradient Average (i.e. momentum)

Per-parameter adaptive learning rates (Adagrad)

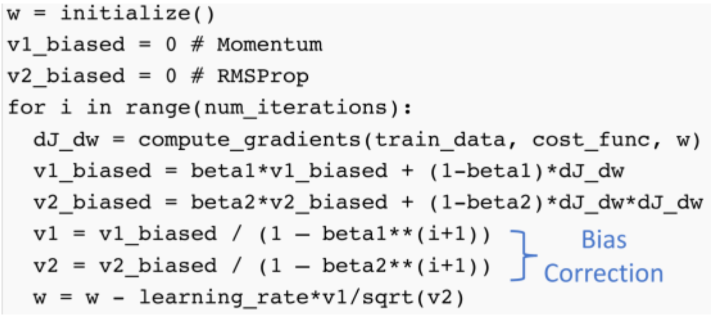
- We have larger steps at steeper areas and smaller steps at shallower areas for gradient descent
- 

  ```
  w = initialize()
  grad_sq = 0
  for i in range(num_iterations):
    dJ_dw = compute_gradients(train_data, cost_func, w)
    grad_sq = grad_sq + dJ_dw * dJ_dw
    w = w - learning_rate*dJ_dw/sqrt(grad_sq)
  ```

  - Keep a separate grad_sq for each parameter
- Intuition
  - Square of gradients focuses on magnitude and not direction
  - Dimensions moving through a region with large gradient will accumulate a larger value into grad_sq, and when you divide by this, you are making the update smaller
    - Dampen
  - Dimensions moving through a region with small gradient will accumulate a smaller value into grad_sq, and when you divide by this, you are making the update larger
    - Accelerate
- Problem
  - No decay of grad_sq, gets bigger and bigger

- Solutions
    - RMSProp
        - Use exponentially weighted average of the square of the gradients
        - 

```
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = beta*grad_sq + (1-beta)*dJ_dw*dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

    - Adam
        - Combines RMSProp and momentum
        - Work well across a wide variety of deep learning problems
        - A good default choice for optimizer
        - 

```
w = initialize()
v1_biased = 0 # Momentum
v2_biased = 0 # RMSProp
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1_biased = beta1*v1_biased + (1-beta1)*dJ_dw
  v2_biased = beta2*v2_biased + (1-beta2)*dJ_dw*dJ_dw
  v1 = v1_biased / (1 – beta1**(i+1))    Bias
  v2 = v2_biased / (1 – beta2**(i+1))    Correction
  w = w - learning_rate*v1/sqrt(v2)
```

Second-order optimization

- Look also at second-order derivative (Hessian)
- Tells about the curvature

Learning rate schedules

- Vary learning rate over training
    - Start high and reduce over time
    - Annealing, decaying the learning rates
- The method in which we decay/anneal the learning rate is referred to as the Decay/Annealing schedule
    - Generally, want to reduce learning rate once progress plateau
- Trade-offs
    - Too slow: wasting time bounding around
    - Too fast: slow down training
- Common decay schedules
    - Step decay
        - Reduce learning rate at fixed points
        - New hyperparameters
            - □ Which intervals to decay
            - □ How to decay at each interval
    - Decay based on function
        - Typically no new hyperparameters needed
        - Exponential decay: $\alpha_t = \alpha_0 e^{-kt}$.
            - □ k is a hyperparameter.
        - Linear decay: $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
            - □ T is the total training iterations.
        - Cosine decay: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos\left(\frac{\pi t}{T}\right)\right)$
        - Inverse sqrt decay: $\alpha_t = \alpha_0 \frac{1}{\sqrt{t}}$
        - 1/t decay: $\alpha_t = \alpha_0 \frac{1}{1+kt}$
- Choosing schedule
    - Try constant learning rate first
    - Step decay: manually decay after progress plateaus
    - Function: non new parameters

Weight initialization

- Hard to start close to a global minima
- Want gradients to be well-behaved (not all zero)
- Initialize with 0 or constants breaks the back propagation
- Initialize with a Gaussian random
    - Breaks symmetry (not all initialized to same value)
    - Mean 0: zero-centered inputs, final weights might be zero-centered
    - Multiplying by x gives the random variable a standard deviation equal to x
    - Good for shallow networks
    - For deeper networks (with large hidden unit) activations get closer to 0
        - Gradient approach 0
        - For tanh, most activations are in saturation
    - Gaussian or uniform
        - Not clear which one is necessarily better
- Xavier initialization
    - Set the variance of Gaussian equal to the number of inputs to the layer
    - For tanh and ReLU
        - `W = (1/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)``
    - For Kaiming/he_normal
        - `W = (2/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)`

Bias initialization

- Simply initialize with 0
    - Symmetry breaking done in initializing the weight parameters
    - Could initialize with small positive number when using ReLU

Data preprocessing

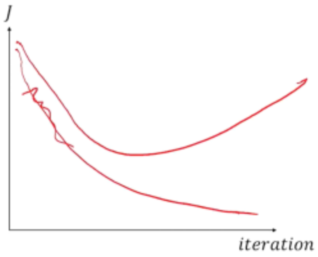
- Consider sigmoid: always positive, parameter updates will be negative
    - Inefficient training
    - Pick a zero-centered activation function
- First layer: if data is all positive, parameter updates will be positive
    - Inefficient training
- Preprocess the input data can help optimization
- Preprocess: Mean subtraction
    - Compute mean for each feature across training samples $\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$
    - Subtract mean from each sample's features $x' = x - \mu$
- Normalization/scaling
    - Compute variance of each feature across all training samples $\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$.
    - Divide each feature by standard deviation $x' = \frac{x}{\sigma}$.
    - Corresponding weights will tend to become similar scale
    - Absolute feature scales
        - Even if all features are on similar scale, we don't want these scales to be large
        - Still leads to large gradients. Small change will lead to big changes in final cost
            - □ Cost is sensitive to small changes to weights
        - Harder to optimize
- Standardization (Z-score normalization)
    - Combine the previous two $x_i' = \frac{x_i - \mu_i}{\sigma_i}$.
- Whitening/decorrelating
- Image data
    - Each pixel is a feature
    - Each feature is on the same scale relative to each other
    - Still need normalization
    - Examples
        - AlexNet: subtracted mean

        - VGGNet: subtracted channel mean
        - ResNet: subtracted channel mean , divided by channel standard deviation
- At inference/prediction
    - Any transformation performed on an input for training must be performed for inputs at prediction

Batch normalization

- Normalizing inputs of the hidden layers
- Stabilizes the optimization problem by giving each layer a target mean and variance
- Makes optimization less sensitive to learning rate and weight initialization
- Algorithm
    - For a given mini-batch with m samples, x is a matrix of shape (n, m).
    - For each input x_i, compute its mean μ_i and variance σ_i^2.
    - For each sample and each feature, normalize $x_i' = \frac{x_i - \mu_i}{\sigma_i}$.
- Zero mean unit variance
    - Too strict, makes optimization problem harder
    - Let the model learn target mean and variance for each layer
- Learned mean and variance
    - Two new trainable parameters γ_i, β_i for each output that act to shift and scale the normalized layer outputs
    - $\widetilde{x_i} = \gamma_i x_i' + \beta_i$.
    - If $\gamma_i = \sigma_i, \beta_i = \mu_i, \widetilde{x_i} = x_i$.
    - If $\gamma_i = 1, \beta_i = 0, \widetilde{x_i} = x_i'$, with zero mean and unit variance.
- Backward propagation
    - $\frac{\partial J}{\partial \beta} = \sum_i \frac{\partial J}{\partial y_i}$.
    - $\frac{\partial J}{\partial \gamma} = \sum_i \frac{\partial J}{\partial y_i} x_i'$.
    - $\frac{\partial J}{\partial x_i'} = \frac{\partial L}{\partial y_i} \gamma$.
    - $\frac{\partial J}{\partial x} = \frac{\gamma}{m\sigma}\left(-\frac{\partial J}{\partial \gamma} x' + m\frac{\partial J}{\partial z} - \frac{\partial J}{\partial \beta}\right)$.
- Can be applied before the nonlinear activation
    - Works well
- Can speed up training
    - Can use larger learning rate
- At prediction
    - Batch norm is a function of all samples in the mini-batch
    - Can't compute mean and variance of only one sample
    - Use moving average
    - Extra processing at inference time
- Slight regularization effect
    - Mean and variance on mini-batch is only an approximation to the actual mean and variance compared to the entire training set activations
    - Introduces noise
    - Unintended regularization effect
- Why
    - Helps stabilize a layer's output
    - Reduces internal covariate shift
    - Smooths the objective landscape
    - Length-direction decoupling

Overfit

- 

- Get more training data
- regularization

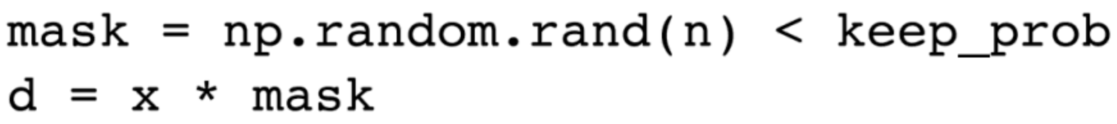
Regularization via cost function

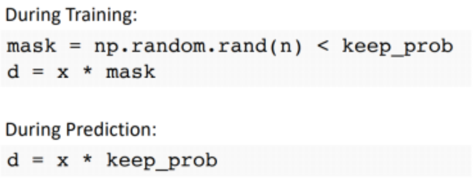
- Add additional terms to encourage regularization in our solution
- $J = \left(\frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y)\right) + R.$
- L2 regularization (weight decay)
  - $J = \left(\frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y)\right) + \sum w^2.$
  - Sum the square of each parameter value
  - Cost can be minimized when each parameter value is small
  - Convex function
  - Global min when all weights are 0
  - Try to minimize the loss and the regularization term
    - Loss term will be large if all weights are zero
  - Specify the importance
    - $J = \left(\frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y)\right) + \lambda \sum w^2.$
    - $\lambda = 0$: we don't optimize for regularization.
    - $\lambda = \infty$: we don't optimize for loss.
    - Default: 0.01
  - Most popular
  - Discourages subset of weights dominating
- L1 regularization
  - $R = \lambda \sum |w|.$
- L2 and L1 (Elastic net)
  - $R = \lambda_{L1} \sum |w| + \lambda_{L2} \sum w^2.$

Regularizing bias parameters

- Not often
- Doesn't have a big impact

Dropout

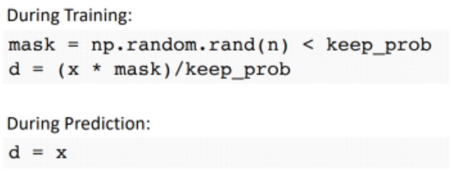
- On each parameter update iteration, randomly remove some hidden unit from the network
- Train a bunch of smaller simpler models and ensemble them together
  - Each model overfits in different ways so averages out
- Don't put too much weight into any particular feature
  - Similar effect to L2 regularization
- Force each unit to learn to work well with a random subset of input units
  - Learn useful features on its own instead of relying on certain input
- Implement dropout by outputting 0 at appropriate locations
  - 
    ```
    mask = np.random.rand(n) < keep_prob
    d = x * mask
    ```
  - Random mask generated on each forward pass
  - Keep_prob is the probability of not dropping a node
  - d is the output with some nodes changed to 0.
  - 2^n unique masks.
- At prediction

    - Non deterministic predictions
    - Expected output value: $E(d) = \sum_{i=1}^{2^n} p(mask_i) d_i(x, mask_i)$.
        - $d_i(x, mask_i)$: output for one mask
        - Each mask occur with $p(mask_i)$
    - Not feasible to compute for any moderate sized layer
    - Good approximation: scale the inputs with keep_prob.
        - 

          During Training:

          ```
          mask = np.random.rand(n) < keep_prob
          d = x * mask
          ```

          During Prediction:

          ```
          d = x * keep_prob
          ```
        - Backward: $\frac{dJ}{dx} = \frac{dJ}{dd} \times mask$
- Inverted dropout
    - 

      During Training:

      ```
      mask = np.random.rand(n) < keep_prob
      d = (x * mask)/keep_prob
      ```

      During Prediction:

      ```
      d = x
      ```
    - Backward: $\frac{dJ}{dx} = \frac{dJ}{dd} \times mask/keep\_prob$.
- Mainly use with FC layers
    - Prone to overfitting compared to conv layers
- Not used with con layers
    - Conv layers aren't so prone to overfitting because each swatch (convolutional location on input volume) is a separate piece of training data

Drop connect:

- Similar to dropout
- Zero out random weights at training (connections) instead of nodes

Data augmentation

- One way of regularization
    - Avoid overfitting to the original data
- Generate new training data from existing training data
- For images
    - Mirror
    - Rotate
    - Blur
    - Saturation
    - Cropping

Regularization

- Common use: L2
- Large FC layer: dropout
- Don't rely on batch norm
- Data augmentation for images

Hyperparameter tunning

- Hyperparameter: any choice that affects your model architecture or optimization process
    - Architecture
        - Number of layers
        - Number of units/filters per layer
    - Optimization
        - Learning rate
        - Weight initialization
        - Optimizer hyperparameters

        - Regularization techniques
- Random search is better than grid search
    - Log scale vs linear scale
        - Log scale: Learning rate from 0.0001 to 1
    - Coarse to fine
        - Do hyperparameter search in initial range of hyperparameter values
        - Find the values that minimize the cost
        - Zoom into a tighter region of values around this set of values and repeat search

General advice

- Start by using a small subset of training set and get the model to 100% accuracy
    - Turn off regularization
    - Flush out buds in optimization flow and glares deficiencies
- Use full training set, find a learning rate that shows good decrease in cost
    - Turn on regularization
    - Can see effect of learning rate in small number of training iterations
- Hyperparameter search
- Monitor histograms of gradients, parameters, activations during training
    - Tensor board
- Get training accuracy high first
    - Low training accuracy means unable to learn
    - Validation accuracy can't do better
- Then work on closing the gap and improve validation accuracy
- Look at failing cases
    - Visualize data
    - Look for patterns
- Look at cost curves
    - Learning rate too big
    - Bad initialization
    - Loss plateaus
    - Decayed learning rate too soon
    - Overfitting
    - Potential underfitting

Transfer learning

- Take a model that was trained for one task and repurpose it for a second similar task
- When repurposing, keep some of the learnings from the first task
- Usage
    - Image
        - Start with CNN trained on a large data set
            - □ We expect this to have learned many important feature
            - □ Early layers of CNNs learn a vocabulary of visual constructs (edges, textures, patterns), no need to relearn
        - Replace output layer with the new output layer
        - Train with new data set, but only update the new output layer's parameters
            - □ Can also let the last couple conv layers be retrained
    - Text and speech
- When
    - Both tasks have same input (images, audio, language data)
    - Significant less training data available for the new task
    - Expect low-level features to be similar in both tasks
- Benefits
    - Leverage previous training efforts so don't need to start from scratch
    - Start with very good parameter values
        - Lower loss
    - Don't need to relearn common low-level features

- Can train a good model even if we have few data

RNN, NLP

October 27, 2021 5:26 PM

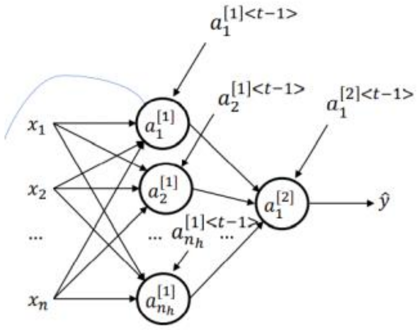
Intro

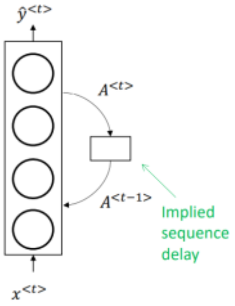
- Simple ML to create approximation for translation does not produce high quality result
- In real world data unfolds over time
  - Information in both individual components of the data and their ordering with respect to other components
  - Need to consider the context

Add context to ML system

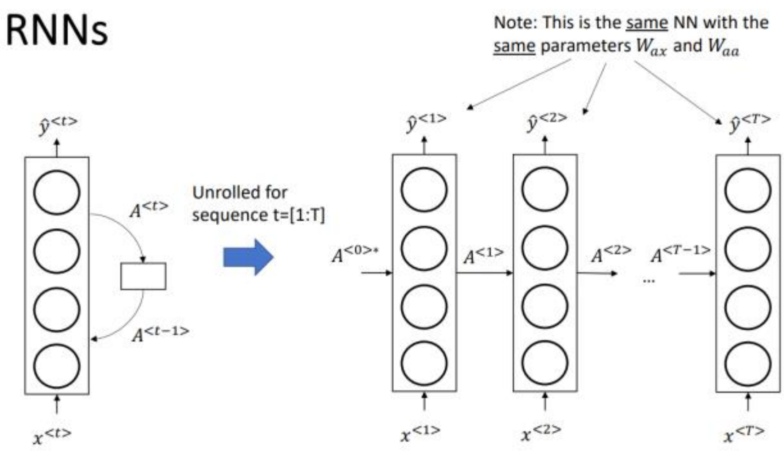
- Can try to increase the inputs to system to reflect the context
  - $y^{<t>} = f\left(x^{<t>}, x^{<t-1>}, \dots, x^{<t-n>}\right).$
  - $x^{<t-1>}, \dots, x^{<t-n>}$ are all the data from the past.
  - Won't scale
- Use activations from the previous step in the sequence can be used to bias the activations on the next step
  - Can simultaneously learn the amount of context required while we learn the input to output mappings

Recurrent Neural Networks (RNNs)

- 

- 

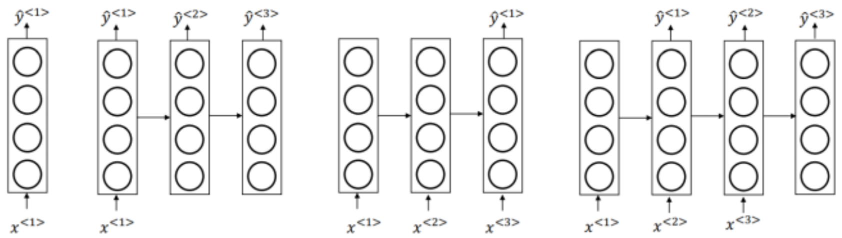
- $a_1^{[1]} = g\left(w_{ax}X_1 + w_{aa}a_1^{[1]<t-1>} + b\right).$
  - $w_{ax}X_1$ is the contribution from current input
  - w_{ax} is regular NN parameters
  - $w_{aa}a_1^{[1]<t-1>}$ is the contribution from current context (previous inputs over time)
  - w_{aa} is previous activation parameters

- 

  RNNs

  Note: This is the same NN with the same parameters W_{ax} and W_{aa}

  Unrolled for sequence t=[1:T]

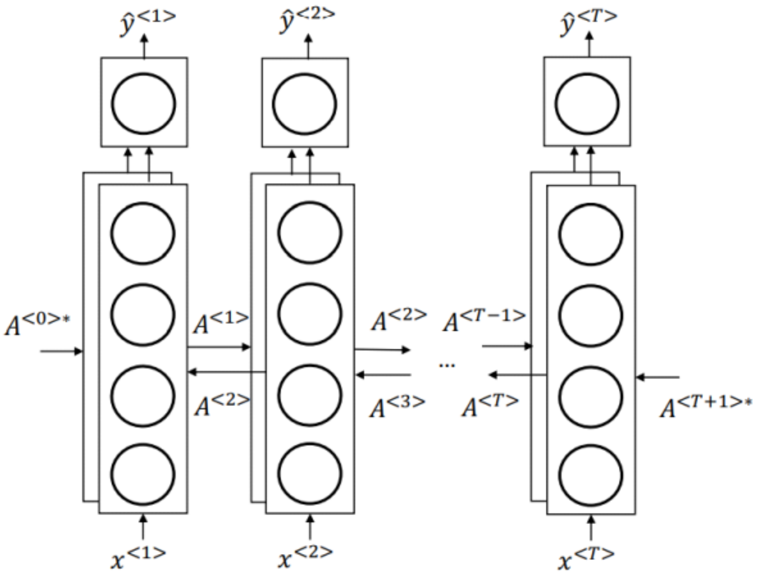
- I/O sequence length flexibility
  - 

    $\hat{y}^{<1>}$
$x^{<1>}$
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<3>}$
$x^{<1>}$
$\hat{y}^{<1>}$
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<3>}$
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$
  - One to one: image classification
  - One to many: image captioning
  - Many to one: sentiment classification
  - Many to many: machine translation
    - Can accommodate extra words
    - Need <eos> to tell us when to stop encoding/decoding

RNN feature extraction

- RNN structure does a form of feature extraction
- e.g. extract similar words
- RNNs isolate elements of sequences like convolutional filters isolate regions of an image

Context

- Context doesn't only flow one way
- Once we have the data, we can look forward and backward in time
- Even when we deploy a system, we can buffer the inputs long enough to consider context in two directions
- Bidirectional RNN
  - 

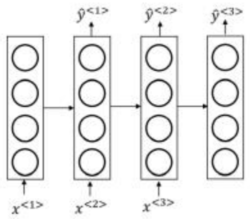
        - $A^{<0>}$ and $A^{<T+1>}$ are set to 0.
    - Forward + backward
    - Combine the outputs
    - Using BRNNs with each sentence considered a sequence is the current state of the art for most NLP applications today

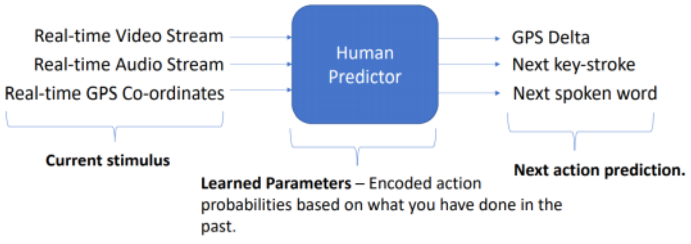
RNN applications

- Sound
- Video
- Natural language
- Online interactions
- Music
- Sports
- Real-time navigating
- Radar tracking

Human behavior prediction

- With what we know, predict what we will do next
- Training and prediction
    - Record everything you see, hear
    - Record everywhere you move and whatever you say and type
    - Train RNN
        - Y = Voice recording, typing record, GPS delta.

          X = Video, Audio recording, GPS location.
    - Deploy
        - 

          Current stimulus

          Next action prediction.

          Learned Parameters – Encoded action probabilities based on what you have done in the past.

          RNN feedback (internal state) – What you have experienced in the recent past (i.e. your *context*)
- Usability
    - Data recording/storage is easy
    - The biggest distributed RNNs would be able to process the data without much of a challenge
    - The only real question would be how predictable are you and would it be worth the time and effort to do that training
- Many human behaviors are predictable and there is a huge money motivation

RNN Notation

- Inputs: $x^{(i)<t>}$ where $i = 1{:}m$ and $t = 1{:}T_x^{(i)}$
- Outputs: $y^{(i)<t>}$ where $i = 1{:}m$ and $t = 1{:}T_y^{(i)}$
- m training examples.
- Each input and output in the training example has a sequence length T.

NLP word representation

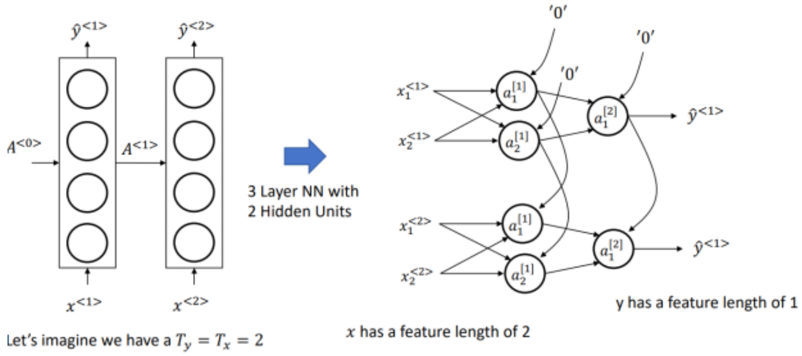
- A standard AI network can only accept numbers as inputs and outputs
- Need to assign each word a number
- Dictionary (vocabulary)

    - Create an ordered dictionary and assign each word number based on its position in the sequence
    - Makes learning task hard and added un-intentioned bias
        - Words are biased together based on their position in the alphabet
- Normalized and less compressed representation
    - One-hot encoding
        - A vector marks which word it is and which word it is not
        - No order bias, better activations
- Unknown words
    - Create one more vector element as unknown word (UKW)
    - Can allow UKW as an output if it makes sense
    - As long as the vocabulary includes all the words that are important for NLP task, should be no problem mapping some words to UKW

RNN loss function

- Expand a single loss function over the entire output sequence
- Define the overall loss to be the sum $L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$.
- With one hot encoding $L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})$.

RNN computation graph and back propagation

- 

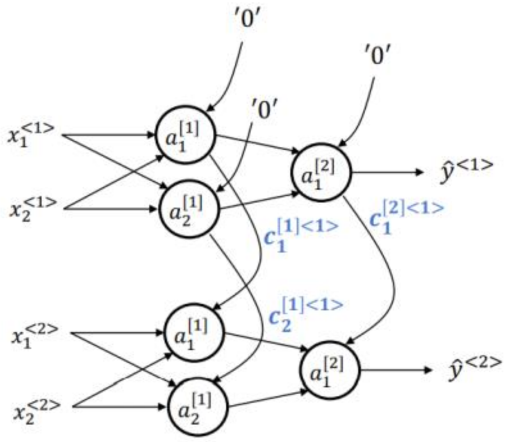
- Step 1. calculate $\hat{y}$ using computation graph.
- Step 2. determine the loss
- Step 3. update each parameter
    - Later values have impact on previous layers.
    - $\frac{\partial \hat{y}^{<2>}}{\partial x_2^{<1>}} = \frac{\partial \hat{y}^{<2>}}{\partial a_1^{[2]<2>}} \cdot \frac{\partial a_1^{[2]<2>}}{\partial a_2^{[1]<2>}} \cdot \frac{\partial a_2^{[1]<2>}}{\partial a_2^{[1]<1>}} \cdot \frac{\partial a_2^{[1]<1>}}{\partial x_2^{<1>}}$
- Step 4. repeat until $J < target$.
- Note: the RNN parameters are being updated with the average gradients on each sample

Vanishing gradients

- As sequence get long, it can be difficult to enable earlier elements to correctly influence later outputs
- We can bypass some activations by holding the previous value
- Gated recurrent unit (GRU)
    - Gating function $\Gamma_\mu = \sigma\left(w_{\mu x} X_1 + w_{\mu a} a_1^{[1]<t-1>} + b_\mu\right)$.
    - Gives value between 0 and 1 based on learned parameters and standard RNN unit inputs
    - Can use the following to decide if we should keep the previous activation or update it
        - $a_1^{[1]<t>} = \Gamma_\mu \widetilde{a_1}^{[1]<t>} + \left(1 - \Gamma_\mu\right) a_1^{[1]<t-1>}$.
        - Standard activation becomes a candidate $\widetilde{a_1}^{[1]<t>} = g\left(w_{ax} X_1 + w_{aa} a_1^{[1]<t-1>} + b\right)$.

Long short term memory (LSTM)

- Most RNNs use the general LSTM to manage the vanishing gradient problems
- Three independent learned functions

    - Update: $\Gamma_\mu = \sigma\left(w_{\mu x}X_1 + w_{\mu a}a_1^{[1]<t-1>} + b_\mu\right)$
    - Forget: $\Gamma_f = \sigma\left(w_{fx}X_1 + w_{fa}a_1^{[1]<t-1>} + b_f\right)$
    - Output: $\Gamma_o = \sigma\left(w_{ox}X_1 + w_{oa}a_1^{[1]<t-1>} + b_o\right)$
- Candidate memory:
    - $\tilde{c}_1^{[1]<t>} = g\left(w_{ax}X_1 + w_{aa}a_1^{[1]<t-1>} + b\right)$.
    - $c_1^{[1]<t>} = \Gamma_\mu \tilde{c}_1^{[1]<t>} + \Gamma_f c_1^{[1]<t-1>}$ (Update the internal memory with both updating and forgetting)
    - Output: $a_1^{[1]<t>} = \Gamma_0 \tanh c_1^{[1]<t>}$.
- 

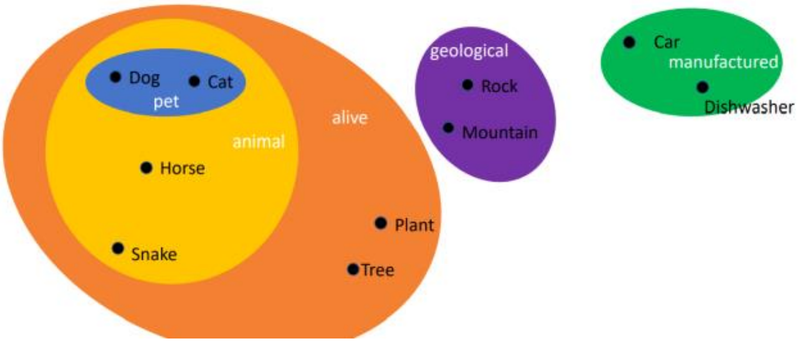
Note: GRU and LSTM are important to RNNs, especially NLP applications

- Structure of sequential data sets
- A key element is critical for a period of time, and then no longer relevant

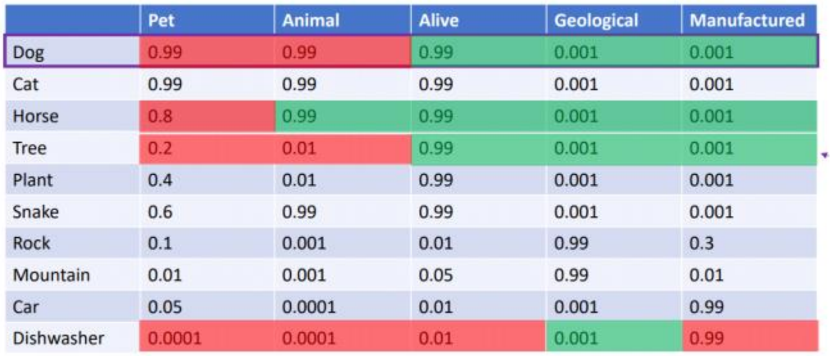
Categorical vs. Binary Cross Entropy

- Softmax: classes are mutually exclusive
- Sigmoid:
    - Classes may overlap, so that case must be interpreted
    - For NLP, overlap could equal UKW
- Depends on the goal of the learning system

Word encodings

- Some words are related
- Closeness map:
    - 

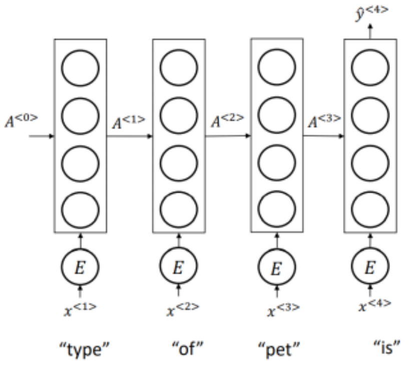
    - Machines can learn these ideas
    - Instead of using a one-hot-encoding for each of the words in the vocabulary, we can imagine that for each word we have a vector where each element of the vector can be thought of as an attribute

    - 

	Pet	Animal	Alive	Geological	Manufactured
Dog	0.99	0.99	0.99	0.001	0.001
Cat	0.99	0.99	0.99	0.001	0.001
Horse	0.8	0.99	0.99	0.001	0.001
Tree	0.2	0.01	0.99	0.001	0.001
Plant	0.4	0.01	0.99	0.001	0.001
Snake	0.6	0.99	0.99	0.001	0.001
Rock	0.1	0.001	0.01	0.99	0.3
Mountain	0.01	0.001	0.05	0.99	0.01
Car	0.05	0.0001	0.01	0.001	0.99
Dishwasher	0.0001	0.0001	0.01	0.001	0.99

    - We can then build an implicit distance between different words and learn the attribute groups
- Embedding matrix
    - Pick the number of attributes (hyperparameter) that we think we will be sufficient to hold our encodings
    - With A attributes and W words in the vocabulary, the embedding matrix E will be of size (A, W).
    - Learning:
        - Algorithms: Word2Vec, negative sampling, GloVe
        - Treat the elements of the matrix as parameters to be learned and use gradient descent to find a good solution

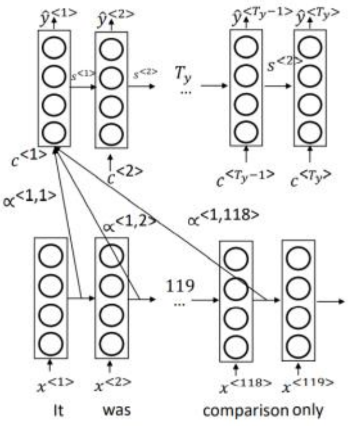
Language models

- Used to predict language based on current and previous inputs (context)
- An encoding that allows similar objects to be represented as similar would make the problem easier
- With the embedding matrix, we can use the one-hot-encoding for each word to extract the vector for the specific word
    - Let one hot be: o_v.
    - v is the position of the 1 in the one-hot vector, then $E \cdot o_v = e_v$.
    - e_v is the encoding of the v^{th} word in the vocabulary.
- Basic language model
    - Over a large set of training data, we would learn to predict the next word from the previous words
    - Normally the inputs would be one-hot-encodings with length equal to the vocabulary size
- Adding learnable embedding matrix
    - By simply multiplying the input one-hot-vector by the embedding matrix
    - 

    - Learned parameters: W_{ax}, W_{aa}, E.
- The embedding matrix can be reused for other applications. If we create E once on a very large and high-quality data set, we can use it as a starting point for other NLP tasks where we have less example data
    - New applications do not have to start from scratch

Attention models

- For the simple machine translation model, the entire sentence must be encoded
- We would like the output sequence generator to pay attention to a selection of the activations of the input words.

- Model that enables the view
  - 

  - Define α as the amount of attention that should be paid to each activation and define $\sum_{t'=1}^{T_x} \alpha^{<1,t'>} = 1$.
    - Computing attention weights could be similar to softmax $\alpha^{<t,t'>} = \frac{\exp\left(e^{<t,t'>}\right)}{\sum_1^{T_x} \exp\left(e^{<t,t'>}\right)}$.
    - But, $e^{<t,t'>}$ can be learned from a neural network.
  - Context for each output sequence $c^{<i>} = \sum_{t'=1}^{T_x} \alpha^{<i,t'>} a^{<t'>}$.