CS231n

Convolutional Neural Networks for Visual Recognition

Sungchan Yi

2019 Fall

Contents

1 Introduction

1.1 History of Computer Vision

- ...
- ImageNet: Image classification challenge
- Huge improvement on 2012 - Convolutional Neural Networks (**CNN**)

1.2 CS231n Overview

- Focus on **image classification**
- Object detection, action classification, image captioning ...
- **CNN** !

2 Image Classification

2.1 Image Classification

- Map: Input Image $\rightarrow$ Predetermined Category
- Semantic Gap - computers only see the pixel values of the image
- Many problems that cause the pixel values to change
  - Viewpoints
  - Illumination
  - Deformation (poses)
  - Occlusion (part of the object)
  - Background Clutter (object looks similar to the background)
  - Interclass Variation (objects come in different sizes and shapes)
- Image classification algorithms must be able to handle all these problems

2.2 Attempts to Classify Images

- Finding corners or edges to determine the object
  - Brittle, doesn't work very well
  - **Not scalable** to work with other objects
- **Data-Driven Approach**

    1. Collect a dataset of images and labels
    2. Use ML algorithms to train a classifier
    3. Evaluate the classifier on new images

- Data-driven approach is much more general

2.3 Nearest Neighbor

1. Memorize all data and labels
2. Predict the label of the **most similar** training image

- How to compare images? **Distance Metric** !
  - L_1 distance (Manhattan Distance)[1]

$$d_1(I_1, I_2) = \sum_{\text{pixel } p} |I_1^p - I_2^p|$$

- With N examples, training takes $\mathcal{O}(1)$, prediction takes $\mathcal{O}(N)$.
- Generally, we want prediction to be fast, but slow training time is OK
- Only looking at a **single** neighbor may cause problems (outliers, noisy data, etc.)
- Motivation for **kNN**s

2.4 k-Nearest Neighbors

- Take the **majority vote** from k closest points
- L_2 distance (Euclidean Distance)

$$d_2(I_1, I_2) = \sqrt{\sum_{\text{pixel } p} (I_1^p - I_2^p)^2}$$

- L_1 distance depends on the coordinate system L_2 doesn't matter
- Generalizing the distance metric can lead to classifying other objects such as texts

2.5 Hyperparameters

- **Hyperparameters** are choices about the algorithm that we set, rather than by learning the parameter from data
- In kNNs, the value of k and the distance metric are hyperparameters
- **Problem dependent**. Must go through trial and error to choose the best hyperparameter

[1]Dumb way to compare, but does reasonable things *sometimes* ...

- Setting Hyperparameters
  - Split the data into 3 sets. Training set, validation set, and test set
  - Train the algorithm with many different hyperparameters on the training set
  - Evaluate with validation set, choose the best hyperparameter from the results
  - Run it once on the test set, and this result goes on the paper
- Never used on image data ...
  - Slow on prediction
  - Distance metrics on pixels are not informative (Distance metric may not capture the difference between images)
  - Curse of dimensionality - data points increase exponentially as dimension increases, but there may not be enough data to cover the area densely (Need to densely cover the regions of the n-dimensional space, for the kNNs to work well)

2.6 Linear Classification

- Lego blocks - different components, as building blocks of neural networks
- **Parametric Approach**
  - Image x, weight parameters W for each pixel in the image
  - A function $f : (x, W) \rightarrow$ numbers giving class scores for each class
  - If input x is an $n \times 1$ vector and there were c classes, W must be $c \times n$ matrix
- Classifier summarizes our knowledge on the data and store it inside the parameter W
- At test time, we use the parameter W to predict
- How to come up with the right structure for f ?
- **Linear** Classifier : $f(x, W) = Wx\ (+b)$
- There may be a bias term b to show preference for some class label
- (Idea) Each row of W will work as a template for matching the input image, and the dot product of each row and the input image vector will give the **similarity** of the input data to the class
- Problems with linear classifier
  - Learning single template for each class
  - If the class has variations on how the class might appear, the classifier averages out all the variations and tries to recognize the object by a single template
  - Using deeper models to remove this restriction will lead to better accuracy

- Linear classifiers draws hyperplanes on the n-dimensional space to classify images
- If hyperplanes cannot be drawn on the space, the linear classifier may struggle (parity problem, multi-modal data)

3 Loss Functions and Optimization

3.1 Motivation

- We saw that W can act as a template for each class
- How do we choose such W ?
- To choose the best W, we should be able to **quantify** the goodness of prediction across the training data

3.2 Loss Functions

- Dataset of examples: $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i is an input data, and $y_i \in \mathbb{Z}$ is a correct label for the data
- Suppose we have a prediction function f, then the **loss over the dataset** is a sum of loss over examples
$$L = \frac{1}{N}\sum_i L_i\big(f(x_i, W), y_i\big)$$
- Now we want to choose a W that **minimizes the loss function**
- **Multiclass SVM loss**
  - Scores vector $s = f(x_i, W)$
  - SVM loss for each data (Hinge loss)
  $$L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)$$
  - As the score for the true category (s_{y_i}) increases, the loss goes down linearly, until it gets above a safety margin because now the example is correctly classified
  - Simply put - We are happy[1] if *the score for the correct label is much higher than all the other scores* by some margin[2]
  - Minimum loss is 0, maximum loss is ∞
  - Quadratic hinge loss - May be used to put more loss on scores that are totally off (Depends on how we want to weigh off different mistakes that the classifier makes)

[1]We are happy when the loss is small

[2]The constant 1 in the equation can actually be generalized

- Suppose we found a W that makes $L = 0$. But this W is not unique[3], so how do we choose such W ?
- We have only written down loss **in terms of the data**. We only told the classifier to find the W that fits the data
- But in practice, we only care about the performance on the test data

3.3 Regularization

- We add an additional **regularization** term to the loss function, which **encourages** the model to somehow pick a simpler W
- We use a regularization penalty(loss) $R(W)$, then

$$L = \frac{1}{N}\sum_i L_i\big(f(x_i, W), y_i\big) + \lambda R(W)$$

  and the λ is a hyperparameter that trades off between data loss and regularization loss
- Common regularization functions
  - L_2 regularization

$$R(W) = \sum_i \sum_j W_{ij}^2$$

  - L_1 regularization

$$R(W) = \sum_i \sum_j |W_{ij}|$$

- Anything that you do to the model that *penalizes the complexity of the model*
- L_1 regularization thinks less non-zero entries are less complex, L_2 thinks spreading numbers all across entries of W is less complex

3.4 Softmax Loss

- Multinomial Logistic Regression
- Multiclass SVM - no interpretation for each score
- Consider the scores as *unnormalized* $\log$ *probabilities of the classes*
- $s = f(x_i, W)$

$$\mathrm{P}\left(Y = k \,|\, X = x_i\right) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{(softmax function)}$$

- Now we have a probability distribution, and we want this to match the distribution that put all it's weight on the correct label

[3] $2W$ will also give 0 loss

- Loss is the $-\log$ of the probability of the true class[4] (**Cross-Entropy**)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

- Minimum loss is 0, maximum loss is ∞

3.5 Optimization

- Minimize the loss function!
- Strategy $\#1$: Random Search (...)
- Strategy $\#2$: **Follow the slope** - Use the geometry of the space to find *which direction from here will decrease the loss function*
- **Gradients** ! - Directional derivatives to find out which direction we should take a step, to minimize the loss function
- Use calculus to compute analytic gradients and use them in code[5]

3.6 Gradient Descent

- The direction of the negative gradient is the fastest decrease
- Key Idea: Starting from the initial guess x_0, iteratively compute x_n by the following

$$x_{n+1} = x_n - \gamma \cdot \nabla F(x_n)$$

  and hope to converge to some x
- Constant γ is the **step size**, sometimes called the **learning rate**, which is an important hyperparameter
- This is **slow** ... Computing gradients for each iteration is too costly
- **Stochastic Gradient Descent** uses a **minibatch** (subset of training data) to estimate the true gradient

3.7 Image Features

- Feeding raw pixel values don't work well
- Compute various **feature representations** of the image, concatenate them and feed it to the classifier

[4]You can view this as the KL divergence between the target and computed distribution, maximum likelihood estimate

[5]Analytic gradients are fast, exact and error-prone, while numerical one is slow and approximate

- What is the best feature transform? [6]
- Example: Color Histogram, Histogram of Oriented Gradients (Local orientation of edges)

4 Introduction to Neural Networks

4.1 Computational Graphs

- Graph to represent any function, where each node is a function (some operation)
- Advantage - **Backpropagation**: Recursively using the chain rule in order to compute the gradient with respect to every variable in the computational graph

4.2 Backpropagation

- Suppose we have a computational graph for loss function L, and we try to calculate the gradient at some node corresponding to the operation f
- Suppose f takes x, y as inputs and outputs z, i.e. $z = f(x, y)$
- We are given the gradient for z, namely $\frac{\partial L}{\partial z}$. With this we want to calculate $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$
- Use the **chain rule** to get

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} \quad \text{and} \quad \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial y}$$

- Usually operation f is simple, so we can find the local gradients, $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ [1] [2]
- Now $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$ will be used **recursively** for the nodes that produced x, y
- **Sigmoid function**

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  Useful fact is that

$$\frac{d\sigma(x)}{dx} = (1 - \sigma(x)) \cdot \sigma(x)$$

- Patterns in backward flow
  - Addition gate *distributes* gradients
  - Max gate *routes* gradients

[6]For instance, conversion from polar to Cartesian coordinates

[1]With this method, we won't have to derive an expression for the gradient for cases where the loss function is very complicated. We can just work on the level where gradients are easy to compute

[2]The expression for $\partial L/\partial z$ may be complex, but in practice, this value will be a single number received from upstream, which we multiply to the local gradient. The expressions for local gradients will be all we need, which will be very simple

  - Multiplication gate *switches* gradients

- Note that multiple gradients coming in from upstream are added together
- When x, y, z become vectors, the gradients will be **Jacobian matrices**
- But computational graphs will be very large, so it is impractical to write down each gradient formula by hand for all parameters

4.3 Neural Networks

- We talked about linear score function $f = Wx$
- 2-layer neural network $f = W_2 \cdot \max(0, W_1 x)$ ...
- Stack multiple layers with non-linear function in between[3]
- **Activation functions**

5 Convolutional Neural Networks

5.1 History

- Mark I Perceptron (1957)
- Adaline / Madaline (1960)
- Backpropagation (1986)
- Restricted Boltzmann Machines (2006)
- Convolutional Neural Networks (2012)

5.2 Structure

- Fully Connected Layer
  - $32 \times 32 \times 3$ image $\rightarrow 3072 \times 1$ vector x
  - 10×3072 weight matrix W
  - Calculate Wx (Each score is the dot product of row of W and x)
- **Convolutional Layer**
  - Keep the structure of the image as $32 \times 32 \times 3$
  - Take a filter, ex. $5 \times 5 \times 3$ filter w

[3]Composition of linear functions will simplify to a linear function

- Slide over the image spatially, and compute dot products (**convolution**) $w^T x + b$ [1]
- Filters *always* extend the full depth of the input volume
- Result: $28 \times 28 \times 1$ **activation map**
- Take **multiple filters** and create a set of activation maps, ex. 6 filters will give a new image of activation maps with size $28 \times 28 \times 6$
- ConvNet is a sequence of convolution layers, interspersed with activation functions
- **Hierarching of filters** - filters at the earlier layers represent low level features (ex. edges)
- At the mid-level, the filters look for more complex features like corners and blobs
- When many filters are stacked, they learn types of simple to more complex features
- Whole structure: Conv, Non-linear (ReLU ...), Conv, Non-linear, Pool layer (once in a while to downsample the size), Conv ... and a fully connected layer at the end to compute the scores for each class

- Spatial Dimensions
  - $N \times N$ image, $F \times F$ filter
  - With stride 1, results in $(N - F + 1) \times (N - F + 1)$ activation map
  - With stride S, size will be $(N - F)/S + 1$
  - If S doesn't divide $N - F$, pad the input image with zeros
  - Common to see Conv layers with stride 1, filters of size $F \times F$ and zero-padding with $\frac{F-1}{2}$ (This will preserve size spatially)
  - Images convolved repeatedly with filters will shrink the image sizes, but shrinking too fast isn't good (won't work well - you lose information)

- Dimensions Summary - The Conv Layer
  - Accepts a volume of size $W_1 \times H_1 \times D_1$
  - With 4 hyperparameters
    - K: Number of filters
    - F: size of filter
    - S: stride
    - P: amount of zero padding
  - Produces a volume of size $W_2 \times H_2 \times D_2$, where

$$W_2 = \frac{W_1 - F + 2P}{S} + 1 \qquad H_2 = \frac{H_1 - F + 2P}{S} + 1 \qquad D_2 = K$$

  - With parameter sharing it produces $F^2 D_1$ weights per filter, for a total of $F^2 D_1 K$ weights and K biases

[1] Consider the filter weights as stretched into a long vector

- 1 number for a single convolution (dot product)
- Activation map is a sheet of neuron outputs, each connected to a small region (receptive field) in the input, all of the sharing parameters (in the filter w)
- With k filters, Conv layer consists of neurons arranged in a 3D grid, and there will be k different neurons all looking at the same region in the input volume
- Note that the fully connected layer will look at the full input volume

5.3 Pooling Layer

- Makes the representations smaller and more manageable (downsampling)
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- With 2 hyperparameters
  - F: spatial extent
  - S: stride
- Produces a volume of size $W_2 \times H_2 \times D_2$ where

$$W_2 = \frac{W_1 - F}{S} + 1 \qquad H_2 = \frac{H_1 - F}{S} + 1 \qquad D_2 = D_1$$

- Doesn't do anything to the depth
- Introduces zero parameters since it computes a **fixed function** of the input, ex. Max, Average
- Uncommon to use zero-padding for pooling layers

6 Training Neural Networks I

6.1 Activation Functions

- **Sigmoid function**
  - Form:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
  - Squashes numbers to range $[0, 1]$
  - Nice interpretation as a saturating firing rate of a neuron
  - Saturated neurons *kill* the gradients - gradient is smaller as x gets further from 0
  - Sigmoid outputs are not zero-centered - if the input to a neuron is always positive, the gradients of the weights will be either all positive or all negative, resulting in increasing or decreasing all the weights. This is inefficient

  - $\exp(x)$ is computationally expensive

- **Hyperbolic Tangent** - $\tanh(x)$
  - Squashes numbers to range $[-1, 1]$
  - Zero centered
  - But kills gradients when saturated
- **Rectified Linear Unit - ReLU**
  - Form: $f(x) = \max\{0, x\}$
  - Does not saturated in the positive region
  - Computationally efficient
  - Converges much faster than $\sigma(x)$, $\tanh(x)$ in practice
  - More biologically plausible than $\sigma(x)$
  - Not zero-centered
  - Gradient is 0 when $x < 0$ - 0 gradient will never update the weights
- **Leaky ReLU**
  - Form: $f(x) = \max\{0.01x, x\}$
  - All benefits of ReLU
  - Gradient will not die!
  - Generalization - **Parametric Rectifier (PReLU)** where $f(x) = \max\{\alpha x, x\}$ [1]
- **Exponential Linear Units - ELU**
  - Form:
    $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$
  - All benefits of ReLU
  - Closer to zero mean outputs
  - Negative saturation regime compared with Leaky ReLU adds some robustness to noise
  - Computation requires $\exp(x)$
- **Maxout Neuron**
  - Form: $f(x) = \max\{w_1^T x + b_1, w_2^T x + b_2\}$
  - Does not have the basic form of dot product (non-linearity)
  - Generalizes ReLU and Leaky ReLU
  - Linear regime, does not saturate, does not die
  - But doubles the number of parameters per neuron

[1] Backpropagate into α (parameter)

6.2 Data Preprocessing

- Preprocess the data (zero centered - gradient problem, normalized, PCA, Whitening)
- Ex. subtract the mean image from all images, subtract per-channel (RGB) mean

6.3 Weight Initialization

- Initializing as small random numbers sampled from Gaussian distribution $\mathcal{N}(0, 0.01^2)$
- This works okay for small networks ... All activations become 0, weights will get small gradients and will update slowly
- What if sampled from $\mathcal{N}(0, 1^2)$ ... Almost all neurons are completely saturated, gradients will be all zero
- Xavier Initialization - sample from the Gaussian distribution and scale it by the size of input $(1/\sqrt{n})$

6.4 Batch Normalization

- To get Gaussian activations!
- Consider a batch of activations at some layer: To make each dimension unit Gaussian, apply
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}\left(x^{(k)}\right)}{\sqrt{\mathrm{Var}\left(x^{(k)}\right)}}$$
- Compute the empirical mean and variance independently for each dimension and normalize
- Can use $\beta^{(k)} = \mathrm{E}\left(x^{(k)}\right)$, $\gamma^{(k)} = \sqrt{\mathrm{Var}\left(x^{(k)}\right)}$ to recover the data, and use these as parameters to be learned (scale and shift)
- Usually inserted after fully connected or convolutional layers and before non-linearity
- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization, slightly reduces the need for dropout (maybe)

6.5 Babysitting the Learning Process

- Preprocess, then choose the architecture
- Double check that the loss function is reasonable
- Start off with small training data (make sure that you can overfit a small portion of the training data - sanity check)
- Start with small regularization and find the learning rate that makes the loss go down

6.6 Hyperparameter Optimization

- Cross-validation strategy
- Coarse stage - only a few epochs to get a rough idea of what parameters work
- Finer stage - longer running time for a finer search
- Monitor and visualize the loss curve and the accuracy (overfitting)
- Track the ratio of weight updates / weight magnitudes (want this to be somewhere around 0.001)

7 Training Neural Networks II

7.1 Fancier Optimization

- Problems with Stochastic Gradient Descent
  - High *condition number* - ratio of largest to smallest singular value of the Hessian matrix
  - Loss function may have a local minima or a saddle point[1]
  - The gradients come from minibatches, so they can be noisy
- **Stochastic Gradient Descent**

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

- SGD + **Momentum**

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

  where v_t represents the 'velocity'[2][3], ρ represents 'friction'

- Gradient may be small near the saddle points / local minima, but the velocity will not be 0, letting us to keep moving and find the optimal parameter
- **AdaGrad**
  - Keep the squares of gradients

$$g_t = g_{t-1} + (\nabla f(x_t))^2$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{g_t + \epsilon}}$$

[1]Also, gradient is very small near the saddle point, which means that we will have slow progress

[2]Initial velocity can be set to 0

[3]The velocity can be interpreted as the weighted sum of recent gradients, where the weights are exponentially decaying

  - Add element-wise scaling of the gradient based on the historical sum of squares in each dimension
  - Over long time, the squared sum of gradients get larger, so the step size gets smaller. This is a feature when the search space is convex (convergence)

- **RMSProp**
  - Weights on the squared sum of gradients

$$g_t = \gamma g_{t-1} + (1-\gamma)\left(\nabla f(x_t)\right)^2$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{g_t + \epsilon}}$$

  - Momentum on the squared gradients

- **Adam** (Adaptive Moment Estimation)
  - $\approx$ RMSProp + Momentum

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla f(x_t)\right)^2$$

  - But since we initialize v_0, m_0 as 0, we consider them as *biased*, so we add a bias correction term to avoid the problem of taking very large steps at the beginning of training

$$\widetilde{m_t} = \frac{m_t}{1-\beta_1^t} \qquad \widetilde{v_t} = \frac{v_t}{1-\beta_2^t}$$

  - Now we have

$$x_{t+1} = x_t - \alpha \cdot \frac{\widetilde{m_t}}{\sqrt{\widetilde{v_t} + \epsilon}}$$

- All these take the **learning rate** (α) as a hyperparameter
- Learning rate may decay over time
- These algorithms are **first-order optimazation** algorithms
- **Second-Order Optimization**
  - Use gradient and Hessian to form quadratic approximation, and step to the minima of the approximation
  - Taylor expansion

$$J(\theta) \approx J(\theta_0) + (\theta-\theta_0)^T \cdot \nabla J(\theta_0) + \frac{1}{2}(\theta-\theta_0)^T \mathbf{H}(\theta-\theta_0)$$

  - Newton parameter update

$$\theta^* = \theta_0 - \mathbf{H}^{-1}\nabla J(\theta_0)$$

  - No learning rate, no hyperparameters

  - Hessian has $\mathcal{O}(n^2)$ elements, inverting takes $\mathcal{O}(n^3)$ ... Too slow for deep learning

- **Model Ensembles**
  - Optimization was about minimizing the objective function and reducing the training loss
  - But we want to increase the accuracy of the model, i.e. reduce the gap between the train and test error
  - Train many independent models, and average their results at test time[4]

7.2 Regularization

- Prevent our model from fitting the training data too well!
- **Dropout**
  - In each forward pass, **randomly** set some neurons' activation to zero
  - Dropout probability is also a hyperparameter
  - (Hand Waving) Prevents co-adaptation of features, forces the network to have a redundant representation
  - Dropout is training a large ensemble of model within a single model (each binary mask) is one model
  - At test time, multiply the output by the dropout probability
- Common pattern
  - **Training**: Add some kind of randomness $y = f_W(x, z)$
  - **Testing**: Average out randomness (sometimes approximate)

$$y = f(x) = \mathrm{E}_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

- **Data Augmentation** - filp images, color jitter, random crops and scales, and more random transformation

7.3 Transfer Learning

- Train the model on a huge dataset, and freeze its weights
- Add/remove custom layers that you want to classify and train only for this layer (**fine tuning**)

[4]May reduce overfitting

8 Deep Learning Software

8.1 CPU & GPU

- CPU: Central Processing Unit
- GPU: Graphical Processing Unit
- CPUs have fewer cores, but each core ia much faster and capable, great at **sequential tasks**
- GPUs have more cores, but each core is much slower, great at **parallel tasks**
- GPUs are great for matrix multiplication
- NVIDIA CUDA
- OpenCL

8.2 Deep Learning Frameworks

- Easy to build big computational graphs
- Easy to compute gradients in computational graphs
- Easy to run it all efficiently on GPU

Deep learning softwares change fast thus they are omitted here.

9 CNN Architectures

9.1 Case Studies

- AlexNet
- ZFNet
- VGGNet
- GoogLeNet
- ResNet

10 Recurrent Neural Networks

10.1 Recurrent Neural Networks

- We might need many-to-many inputs/outputs for our model (variable length processing)

- RNN reads input x and predicts a vector y at some time step
- **Recurrence Formula**
$$h_t = f_W(h_{t-1}, x_t)$$
where h_t is the state at time t, x_t is the input vector at time t, and f_W is some function with parameters W
- Notice that the same function and the same set of parameters are used at every time step, thus the name *recurrent*
- Example
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \qquad y_t = W_{hy}h_t$$
With the output h_t, calculate y_t using weights W_{hy} to compute the prediction vector y_t
- **Truncated Backpropagation** through time - Run forward and backward through chunks of the sequence instead of the whole sequence
- Image Captioning: CNN+RNN (+ Attention: which part of the image to look at, for every time step)
- Backpropagation on RNN: we need to multiply by some part of the weight matrix, *multiple times*, which may cause gradients to either explode or vanish (gradient clipping - scale gradient if its L_2 norm is too big, or change RNN architecture)

10.2 Long Short Term Memory (LSTM)

- h_t: hidden state, c_t: cell state
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g \qquad h_t = o \odot \tanh(c_t)$$
  - f: Forget gate, whether to erase cell
  - i: Input gate, whether to write to cell
  - g: Gate gate, how much to write to cell
  - o: Output gate, how much to reveal cell
- Forget a portion of previous cell state c_{t-1}, and determine whether to write g to the cell state, and squash the cell state using $\tanh$, multiply the output to decide how much we want to reveal the cell
- Backpropagation: Element-wise multiplication by the forget gate (uninterrupted gradient flow)

- **Gated Recurrent Units** (GRU)

$$\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
\widetilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t
\end{aligned}$$