[Notes] Deep Learning Specialization @DeepLearning.AI

Xu, Bangyao
*bangyaoxu*0321@*gmail.com*

December 2, 2020

1 Neural Networks and Deep Learning

1.1 Introduction to deep learning

- Neural Network: Standard NN, Convolutional NN, Recurrent NN
- Structured Data vs. Unstructured Data (Audio, Image, Text)
- Drivers of deep learning progress: Data + Computation + Algorithms

1.2 Neural Networks Basics

- $\mathbf{X} \in \mathbb{R}^{n \times m}$, where n is # of *variables* and m is # of *samples*
- $\mathbf{Y} \in \mathbb{R}^{1 \times m}$, where m is # of *samples*
- Logistic Regression (LR): $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$
- $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)}) \Rightarrow \hat{y}^{(i)} \approx y^{(i)}$
- Loss function: $\mathcal{L}(\hat{y}, y) = -\Big(y \log \hat{y} + (1-y) \log(1-\hat{y})\Big) \Rightarrow$ the error for *a single training example*
- Cost function: $\mathcal{J}(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \Rightarrow$ the average of the loss functions of *the entire training set*
- Gradient Descent: $w = w - \alpha \frac{\partial \mathcal{J}(w,b)}{\partial w}$; $b = b - \alpha \frac{\partial \mathcal{J}(w,b)}{\partial b}$, where α is called the *learning rate*
- Computation graph: back propagation $\Rightarrow$ chain rule in calculus

- Vectorization ⇒ avoid using explicit for-loop whenever possible
- Explanation of LR cost function: $\mathbb{P}(y|x) = \hat{y}^y(1-\hat{y})^{(1-y)} \Rightarrow \log\Big(\mathbb{P}(y|x)\Big) = y\log(\hat{y}) + (1-y)\log(1-\hat{y}) = -\mathcal{L}(\hat{y}, y) \Rightarrow$ maximum likelihood estimation (MLE) $\Rightarrow$ minimize cost function $\mathcal{J}(w, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

1.3 Shallow neural networks

- NN representation: Input layer + Hidden layer + Output layer
- NN computation: $Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]}, A^{[i]} = \sigma(Z^{[i]})$ where $i = 1, 2$
- Vectorized implementation ⇒ stack up training examples horizontally
- Activation functions: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$; $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$; ReLU$(z) = \max(0, z)$; Leaky ReLU$(z) = \max(0.01z, z)$
- A linear hidden layer is more or less useless because the composition of two linear functions is itself a linear function
- Derivatives of the activation functions:
- $\sigma'(z) = \sigma(z)\Big(1 - \sigma(z)\Big)$;
- $\tanh'(z) = 1 - \Big(\tanh(z)\Big)^2$;
- ReLU$'(z) = \mathbb{I}\{z \geq 0\}$;
- Leaky ReLU$'(z) = 0.01 \times \mathbb{I}\{z < 0\} + \mathbb{I}\{z \geq 0\}$
- Random initialization ⇒ initialize weights to very small random values

1.4 Deep Neural Networks

- Notations: $L =$ #layers; $n^{[l]} =$ #units in layer l; $a^{[l]} =$ activations in layer l, where $a^{[l]} = g^{[l]}(z^{[l]})$; $w^{[l]}, b^{[l]} =$ parameters for $z^{[l]}$
- $X = a^{[0]}$ and $\hat{y} = a^{[L]}$
- $Z^{[l]} = W^{[l]}A^{[i-1]} + b^{[l]}$; $A^{[l]} = g^{[l]}(Z^{[l]})$, where $l = 1, 2, ..., L$
- Dimensions of $W^{[l]}$ & $b^{[l]}$: $\dim(W^{[l]}) = (n^{[l]}, n^{[l-1]})$; $\dim(b^{[l]}) = (n^{[l]}, 1)$
- $\dim(Z^{[l]}) = \dim(A^{[l]}) = \dim(dZ^{[l]}) = \dim(dA^{[l]}) = (n^{[l]}, m)$

- There are mathematical functions that are much easier to compute with deep networks than with shallow networks e.g. XOR operations over n samples: $\mathcal{O}(\log(n))$ vs. $\mathcal{O}(2^n)$
- Forward propagation: $a^{[l-1]} \Rightarrow a^{[l]}$, cache $Z^{[l]}, W^{[l]}, b^{[l]}, l = 1, ..., L$
- Backward propagation: $da^{[l]} \Rightarrow da^{[l-1]}$, output $dW^{[l]}, db^{[l]}, l = 1, ..., L$
- Hyperparameters: learning rate α, # iterations, # hidden layer L, # hidden units, choice of activation functions, etc.
- Empirical process: Idea $\Rightarrow$ Code $\Rightarrow$ Experiment $\Rightarrow$ Refined idea

2 Improving Deep Neural Networks: Hyperparameter tuning, Regularization & Optimization

2.1 Practical aspects of Deep Learning

- Data split: train set + dev set + test set (e.g. 98% + 1% + 1%)
- Dev set $\Rightarrow$ cross validation; Test set $\Rightarrow$ unbiased estimate
- Make sure the dev set and test set come from the same distribution
- High bias $\Rightarrow$ under-fitting; High variance $\Rightarrow$ over-fitting
- Train set error $\gg$ Optimal (Bayes) error $\Rightarrow$ high bias issue
- Dev set error $\gg$ Train set error $\Rightarrow$ high variance issue
- High bias solutions: bigger network, run training longer, etc.
- High variance solutions: more data, regularization, etc.
- In modern deep learning, big data era, there's no bias-variance tradeoff
- NN Regularization: $\mathcal{J}(w^{[1]}, b^{[1]}, ..., w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||w^{[l]}||_F^2$, where Frobenius norm $||w^{[l]}||_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$
- L2 norm regularization is also called *weight decay*:
- $w^{[l]} := w^{[l]} - \alpha \times dw^{[l]}_{original} \Rightarrow w^{[l]} := (1 - \frac{\alpha\lambda}{m}) w^{[l]} - \alpha \times dw^{[l]}_{original}$
- $\lambda \uparrow \Rightarrow w^{[l]} \downarrow$: reduce impact of a lot of the hidden units & make the activation functions more linear ($z^{[l]} \downarrow$)

- Dropout regularization: zero out the hidden units with keep-prob (e.g. 0.8) & invert back to remain the expected value the same
- Dropout shrinks the weights and does some of those outer regularization that helps prevent over-fitting
- One big downside of drop out is that the cost function $\mathcal{J}$ is no longer well-defined
- Data augmentation: flip horizontally, zoom in, rotate randomly
- Early stopping: choose optimal #iterations for dev set error (main downside: violate the principle of orthogonalization)
- Use the same μ and σ^2 from training set to normalize test set
- it's important to normalize the features if the input features come from very different scales
- Random weight initialization $\Rightarrow$ solve vanishing or exploding gradients:
- ReLU $\Rightarrow w^{[l]} = randn \times \sqrt{\frac{2}{n^{[l-1]}}}$; tanh $\Rightarrow w^{[l]} = randn \times \sqrt{\frac{1}{n^{[l-1]}}}$
- Gradient check: remember regularization; doesn't work with dropout

2.2 Optimization algorithms

- Mini-batch t: $X^{\{t\}}, Y^{\{t\}} \Rightarrow$ implement one step of gradient descent
- Epoch $\Rightarrow$ a single pass through the whole training set
- Mini-batch size $= m \Rightarrow$ Batch gradient descent (too long per iteration)
- Mini-batch size $= 1 \Rightarrow$ Stochastic gradient descent (lose speedup for vecterization)
- Mini-batch size in-between: fastest learning (make full use of vectorization, make progress without processing entire training set)
- Typical mini-batch size: 64 $(2^6), 128$ $(2^7), 256$ $(2^8), 512$ (2^9)
- Make sure mini-batches $(X^{\{t\}}, Y^{\{t\}})$ fit in CPU/GPU memory
- Exponentially weighted average: $v_t = \beta v_{t-1} + (1-\beta)\theta_t$, $v_0 = 0$
- Bias correction can help to get a better estimate early on

- Basic idea of *gradient descent with momentum* is to compute an exponentially weighted average of gradients and use it to update weights
- Gradient descent with momentum:
- $V_{dW} = \beta v_{dW} + (1-\beta)dW$; $V_{db} = \beta v_{db} + (1-\beta)db$;
- $W = W - \alpha v_{dW}$; $b = b - \alpha v_{db}$, where $\beta = 0.9$
- RMSprop (Root Mean Square prop):
- $S_{dW} = \beta S_{dW} + (1-\beta)dW^2$; $S_{db} = \beta S_{db} + (1-\beta)db^2$;
- $W := W - \alpha \frac{dW}{\sqrt{S_{dW}}+\epsilon}$; $b := b - \alpha \frac{db}{\sqrt{S_{db}}+\epsilon}$, where $\epsilon = 10^{-8}$
- Adam (adaptive moment estimation $\Rightarrow$ momentum + RMSprop):
- $V_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$; $V_{db} = \beta_1 v_{db} + (1-\beta_1)db$;
- $S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2$; $S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$;
- $V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t}$; $V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$;
- $S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}$; $S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$;
- $W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}}+\epsilon}$; $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}}+\epsilon}$;
- where $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ and α needs to be tuned
- Learning rate decay: $\alpha = \frac{1}{1+decay-rate \times epoch\#}\alpha_0$; $\alpha + 0.95^{epoch\#}\alpha_0$; $\alpha = \frac{k}{\sqrt{epoch\#}}\alpha_0$; $\alpha = \frac{k}{\sqrt{t}}\alpha_0$; or discrete staircase and manual decay
- Local optima: high-dimensional spaces $\Rightarrow$ saddle points/plateaus

2.3 Hyperparameter tuning, Batch Normalization & Programming Frameworks

- Hyperparameters: 1) α; 2) $\beta \approx 0.9$, mini-batch size, #hidden units; 3) #layers, learning rate decay; 4) $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$
- Coarse to fine search (zoom in) random values instead of the grid
- Uniformly random search for hyperparameters: linear/log scale
- Panda (babysit one model) vs. Caviar (train some models in parallel)

- Batch Norm (BN): normalize not only input layer but also hidden layer $\Rightarrow \tilde{Z}^{(i)} = \gamma Z^{(i)}_{norm} + \beta$, where $Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$, μ and σ are *mean* and *standard deviation* of $Z^{(i)}$
- There's no point having $b^{[l]}$ since BN zeroes out the mean of Z^l
- Covariate shift $\Rightarrow$ BN reduces the amount that distribution of hidden unit values shifts around $\Rightarrow$ BN causes input values to be more stable
- BN allows each layer of the network to learn by itself, a little bit more independently of other layers $\Rightarrow$ speed up learning in whole network
- BN has a slight regularization effect since the mean & variance are computed on mini-batches, which adds some noise like dropout
- For testing set, estimate μ and σ using exponentially weighted average across mini-batches since it processes a single example at a time
- Softmax activation: $g(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_i e^{x_i}}$, where x_i is the i^{th} element in $\mathbf{x}$
- Softmax regression generalizes logistic regression to $\mathcal{C}$ classes
- Loss function for Softmax: $\mathcal{L}(\hat{y}, y) = -\sum_j y_j \log(\hat{y}_j)$
- TensorFlow have already built-in the necessary backward functions

3 Structuring Machine Learning Projects

3.1 ML Strategy (1)

- Chain of assumptions in ML (orthogonalized controls):
- 1) Fit training set well on cost function $\Rightarrow$ bigger network, Adam;
- 2) Fit dev set well on cost function $\Rightarrow$ regularization, bigger train set;
- 3) Fit test set well on cost function $\Rightarrow$ bigger dev set;
- 4) Perform well in real world $\Rightarrow$ change dev set or cost function
- A single number evaluation metric speeds up the iterative process of improving ML algorithm (e.g. F1 Score vs. Precision & Recall)
- N metrics $\Rightarrow$ 1 Optimizing metric + $(N-1)$ Satisficing metrics

- Choose the dev set and test dev from same distribution to reflect data expected to get in the future and consider important to do well on
- Set the test set to be big enough to give high confidence in the overall performance of the system
- Current evaluation metric is not giving the correct rank order preference ⇒ defining a new evaluation metric
- Bayes optimal error ⇒ the very best theoretical function mapping from $\mathbf{X}$ to y that can never be surpassed
- Reasons for why progress often slows down when surpassing human level performance:
- 1) Human level performance is not that far from Bayes optimal error;
- 2) Some tactics are harder to apply once the algorithm is doing better than humans (e.g. get labeled data from humans)
- Avoidable error = Training error - Human-level error ($\approx$ Bayes error)
- Choice of the human-level error as proxy for Bayes error ⇒ target the focus on bias or variance reduction
- It's a bit harder for computers to surpass human-level performance on *natural perception task*
- Reducing bias ⇒ 1) train bigger model 2) train longer/better optimization algorithms 3) NN architecture/hyperparameters search
- Reducing variance ⇒ 1) more data 2) regularization 3) NN architecture/hyperparameters search

3.2 ML Strategy (2)

- Error analysis: evaluate multiple error categories in parallel ⇒ determine the ceiling of performance ⇒ prioritize/inspire new directions
- DL algorithms are quite robust to random errors (not systematic error) in the training set
- Incorrect labels in the dev set ⇒ include this category in the error analysis ⇒ fix the labels if it's significant

- Setup dev/test set & metric ⇒ build initial system quickly ⇒ use bias/variance analysis & error analysis to prioritize next steps
- Allow training set to come from a different distribution than dev and test set ⇒ have more training data (e.g. pictures from web vs. phone)
- Training-dev set (same distribution as training set, but not used for training) ⇒ separate variance problem and data mismatch problem:
- 1) Training error - Training-dev error ⇒ variance problem;
- 2) Training-dev error - Dev error ⇒ data mismatch problem
- How to address data mismatch: 1) manual error analysis to understand the difference between training and dev/test sets 2) make training data more similar or collect more data similar to dev/test sets ⇒ artificial data synthesis (try to avoid over-fitting)
- Transfer learning: Pre-training + Fine tuning
- Transfer learning makes sense when transferring from $\mathbb{A}$ to $\mathbb{B}$:
- 1) $\mathbb{A}$ and $\mathbb{B}$ have same input $\mathbf{X}$;
- 2) Having more data for $\mathbb{A}$ than $\mathbb{B}$;
- 3) Low level features from $\mathbb{A}$ could be useful for learning $\mathbb{B}$
- Multi-task learning: train one neural network to do several things
- Multi-task learning makes sense when:
- 1) Tasks could benefit from having shared lower-level features;
- 2) Amount of data for each task is quite similar;
- 3) A big enough neural network can do well on all the tasks
- End-to-end deep learning vs. Multiple stages of processing
- Pros of end-to-end deep learning: 1) let the data speak 2) less hand-designing of components needed
- Cons of end-to-end deep learning: 1) need large amount of data 2) excludes potentially useful hand-designing of components

4 Convolutional Neural Networks

4.1 Foundations of Convolutional Neural Networks

- Computer vision problems: 1) image classification 2) object detection 3) neural style transfer
- Edge detection: Convolution operation + Vertical & Horizontal filter
- Convolution operation (shrink problem + throw away information):
- $n \times n$ input $\bigotimes$ $f \times f$ filter $\Rightarrow (n-f+1) \times (n-f+1)$ output
- Padding: $n \times n \Rightarrow (n+2p) \times (n+2p)$
- Valid convolution $\Rightarrow p = 0$
- Same convolution $\Rightarrow p = \frac{f-1}{2} \Rightarrow$ output size = input size
- Stride: $n \times n \Rightarrow \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$
- Convolution on volumes + Multiple filters:
- $n \times n \times n_c \bigotimes f \times f \times n_c \Rightarrow (n-f+1) \times (n-f+1) \times n_c^{'}$
- Convolution layer notations:
- $f^{[l]}$ = filter size; $p^{[l]}$ = padding; $s^{[l]}$ = stride; $n_c^{[l]}$ = number of filters;
- filter: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$; activation: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$;
- weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$; bias: $n_c^{[l]}$
- Last step of a ConvNet: unroll the volume data into a vector and feed it into logistic/softmax $\Rightarrow$ prediction for final output
- Max pooling has no parameters for gradient descent to learn
- Max pooling computation is done independently on each of channels
- Max pooling vs. Average pooling
- As goes deeper in the neural network, usually height and width will decrease, whereas the number of channels will increase
- Advantages of convolutional layers: 1) parameter sharing 2) sparsity of connections

4.2 Deep convolutional models: case studies

- Classic networks: LeNet-5, AlexNet, VGG
- Residual block: $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) \Rightarrow$ Residual network (ResNet)
- For ResNet, even as the number of layers gets deeper, the training error keeps on going down $\Rightarrow$ effective to train very deep networks
- The identity function is easy for residual block to learn $\Rightarrow a^{[l+2]} = a^{[l]}$ due to the *skip connections*: $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(w^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}) = g(a^{[l]}) = a^{[l]}$ with $g(x) =$ ReLU and $w^{[l+2]} = b^{[l+2]} = 0$
- 1×1 convolution: $\Rightarrow$ shrink/keep/increase number of channels
- Inception network: do all kinds of layers and concatenate the outputs
- Bottleneck layer: 1×1 convolution $\Rightarrow$ shrink number of channels $\Rightarrow$ reduce computational costs
- Transfer learning: freeze all of the earlier layers weights $\Rightarrow$ from $\mathbf{X}$ to a feature vector (pre-computed) + only train the last few layers
- Download open source weights & use that as initialization for problem
- Common data augmentation methods: 1) mirroring 2) random cropping 3) rotation 4) shearing 5) local warping
- Color shifting: take different values of RGB to distort color channels
- Data augmentation and training can run in parallel in CPU/GPU
- Hand-engineering (hacks) is the best way to get good performance when data is not as much as needed
- Tips for doing well on benchmarks:
- 1) ensembling $\Rightarrow$ independent networks + average outputs;
- 2) multi-crop at test time $\Rightarrow$ run on multiple versions + average results

4.3 Object detection

- Classification with localization: training set also contains 4 additional numbers (b_x, b_y, b_h, b_w) for the bounding box of the object detected
- Target label $y = [p_c, b_x, b_y, b_h, b_w, c_1, ..., c_N]^T$, where p_c indicates there's any object or not; b_x, b_y, b_h, b_w specify the location of the bounding box; and $c_1, ...c_N$ are N classes (e.g. pedestrian, car, motorcycle)
- Loss function example as the square error:

$$\mathcal{L}(\hat{y}, y) = \begin{cases} (\hat{p_c} - p_c)^2 + (\hat{b_x} - b_x)^2 + ... + (\hat{c_N} - c_N)^2, & \text{if } p_c = 1 \\ (\hat{p_c} - p_c)^2, & \text{if } p_c = 0 \end{cases}$$

- Landmark detection $\Rightarrow$ use the coordinates $(l_{(i,x)}, l_{(i,y)})$ as the label y
- Sliding windows detection $\Rightarrow$ take square boxes, slide across the entire image, classify region with stride as containing the object or not
- Convolution implementation of sliding windows: implement the entire image convolutionally instead of sequentially
- YOLO (You Only Look Once) algorithm: $n \times n$ grid cells $\Rightarrow$ ConvNet outputs the bounding boxes coordinates explicitly $(n \times n \times \dim(y))$
- Intersection over Union (IoU) $= \frac{size\ of\ intersection}{size\ of\ union} \geq 0.5 \Rightarrow$ map localization to the accuracy
- Non-max suppression $\Rightarrow$ output the maximal probabilities classifications but suppress the close-by ones that are non-maximal
- Non-max suppression algorithm:
- 1) Discard all the boxes with lower probabilities (e.g. $p_c \leq 0.6$);
- 2) Pick the box with the largest p_c and output as a prediction;
- 3) Discard any remaining box with IoU ≥ 0.5 with box in step 2)
- 4) Repeat steps 2) & 3) until there's no remaining boxes
- It's right to independently carry out non-max suppression multiple times, one on each of the outputs classes

- Anchor box algorithm ⇒ each object in training image is assigned to grid cell that contains its midpoint and anchor box for the grid cell with the highest IoU ⇒ deal with what happens if two objects appear in the same grid cell & allow the algorithm to specialize better
- Region proposal ⇒ select just a few windows and run the ConvNet classifier on just a few windows (segmentation algorithm)

4.4 Special applications: Face recognition & Neural style transfer

- Face verification (1:1) vs. Face recognition (1:K)
- One-shot learning ⇒ learn from one example (e.g. only have one picture in the employee database) to recognize the person again
- Face verification problem ⇒ $d(img1, img2)$ = degree of difference between images:

$$\begin{cases} Same, & \text{if } d(img1, img2) \leq \tau \\ Different, & \text{if } d(img1, img2) > \tau \end{cases}$$

- Siamese network: run two identical convolutional neural networks on two different inputs ⇒ $d(x^{(i)}, x^{(j)}) = ||f(x^{(i)}) - f(x^{(j)})||_2^2$, where $f(x)$ is fully connected layer that is deeper in the network (encoding of x)
- Triplet loss learning objective: $||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2 \Rightarrow ||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0$, where A, P, N stand for anchor, positive, negative; and α is the margin to avoid trivial solution
- Triplet loss function & cost function:
- $\mathcal{L}(A, P, N) = \max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0)$
- $\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$
- Choose triplets that are hard ($d(A, P) \approx d(A, N)$) to train on since $d(A, P) + \alpha \leq d(A, N)$ is easily satisfied if A, P, N are chosen randomly
- Learning the similarity function: train a pair of images on Siamese network and add the logistic layer in the end to predict similar or dissimilar images ⇒ $\hat{y} = \sigma\big(\sum_k (w_k | f(x^{(i)})_k - f(x^{(j)})_k| + b\big)$
- Neural style transfer: Content (C) + Style (S) = Generated image (G)

- Neural style transfer cost function: $\mathcal{J}(G) = \alpha\, \mathcal{J}_{Content}(C,G) + \beta\, \mathcal{J}_{Style}(S,G)$
- Find the generated image G:
- 1) Initiate G randomly (e.g. white noise pixes);
- 2) Use gradient descent to minimize $\mathcal{J}(G) \Rightarrow G := G - \frac{\partial}{\partial G}\mathcal{J}(G)$
- $\mathcal{J}_{Content}(C,G) = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2$, where $a^{[l](C)}$ and $a^{[l](G)}$ are the activation of layer l of the pre-trained ConvNet (e.g. VGG) on images
- Style is defined as correlation between activations across channels
- Style matrix: $G^{[l]}_{k,k'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=i}^{n_W^{[l]}} a^{[l]}_{i,j,k} a^{[l]}_{i,j,k'}$, where $a^{[l]}_{i,j,k}$ = activation at (i,j,k), $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$, and $k, k^{'} = 1, ..., n_c^{[l]}$
- $\mathcal{J}^{[l]}_{Style}(S,G) = \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_c^{[l]}\right)^2} ||G^{[l](S)} - G^{[l](G)}||^2_F$
  $= \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_c^{[l]}\right)^2} \sum_k \sum_{k'} (G^{[l](S)}_{k,k'} - G^{[l](G)}_{k,k'})^2$
- $\mathcal{J}_{Style}(S,G) = \sum_l \lambda^{[l]} \mathcal{J}^{[l]}_{Style}(S,G)$
- 2D convolution can be generalized to 1D and 3D as well

5 Sequence Models

5.1 Recurrent Neural Networks

- Notations for sequence model:
- $X^{(i)<t>} = t^{th}$ element in the input sequence of training example i;
- $T_x^{(i)}$ = input sequence length for training example i;
- $y^{(i)<t>} = t^{th}$ element in the output sequence of training example i;
- $T_y^{(i)}$ = output sequence length for training example i
- Recurrent Neural Networks (RNN) forward propagation:
- $a^{<t>} = g(w_{aa} a^{<t-1>} + w_{ax} x^{<t>} + b_a)$; $\hat{y}^{<t>} = g(w_{ya} a^{<t>} + b_y)$
- Simplified RNN notation: $w_a = [w_{aa}, w_{ax}]$

- $a^{<t>} = g(w_a[a^{<t-1>}, x^{<t>}] + b_a)$; $\hat{y}^{<t>} = g(w_y a^{<t>} + b_y)$
- Loss function $\Rightarrow$ Back-propagation through time:
- $\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log(\hat{y}^{<t>}) - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})$;
- $\mathcal{L}^{<t>}(\hat{y}, y) = \sum_{t=1}^{T_x} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$
- RNN architectures:
- Many-to-many: $x^{<1>}, ..., x^{<T_x>} \Rightarrow \hat{y}^{<1>}, ..., \hat{y}^{<T_y>}$;
- Many-to-one: $x^{<1>}, ..., x^{<T_x>} \Rightarrow \hat{y}$; (Sentiment classification)
- One-to-one: $x \Rightarrow \hat{y}$; (Standard generic nerual network)
- One-to-many: $x \Rightarrow \hat{y}^{<1>}, ..., \hat{y}^{<T_y>}$ (Sequence generation)
- Language model estimates probability of particular sequence of words
- Language model with RNN (large corpus of text $\Rightarrow$ tokenize):
- $\hat{y}^{<t>} = \mathbb{P}(y^{<t>} | y^{<1>}, ..., y^{<t-1>})$, where $y^{<t>} = x^{<t+1>}$;
- Loss function: $\mathcal{L}(\hat{y}, y) = -\sum_t \sum_i y_i^{<t>} \log(\hat{y}_i{}^{<t>})$
- Word-level language model vs. Character-level language model
- Sampling a sequence from a trained RNN $\Rightarrow$ randomly sample from softmax distribution until the end of sentence (EOD) token
- Basic RNNs tend not to be good at capturing long-range dependencies $\Rightarrow$ vanishing gradients & exploding gradients
- Gradient clipping is a robust solution that takes care of exploding gradients $\Rightarrow$ re-scale some of gradient vectors beyond some threshold
- Gated Recurrent Unit (GRU) $\Rightarrow$ introduce c = memory cell:
- $c^{<t>} = a^{<t>}$; Candidate: $\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$;
- Gate: $\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \Rightarrow$ decide when to update $c^{<t>}$;
- $c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + (1 - \Gamma_u) \times c^{<t-1>}$; $\Gamma_u = 0 \Rightarrow$ don't update $c^{<t>}$
- Full GRU $\Rightarrow$ $\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r \times c^{<t-1>}, x^{<t>}] + b_c)$, where $\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$

- Long Short Term Memory (LSTM):
- $\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$;
- Update: $\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$;
- Forget: $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$;
- Output: $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$;
- $c^{<t>} = \Gamma_u \times \tilde{c}^{<t>} + \Gamma_f \times c^{<t-1>}$
- $a^{<t>} = \Gamma_o \times \tanh(c^{<t>})$
- Bidirectional RNN (BRNN): $\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$
- Deep RNN: $a^{[l]<t>} = g(W_a^{[l]}[a^{[l]<t-1>}, a^{[l-1]<t>}] + b_a^{[l]})$

5.2 Natural Language Processing & Word Embeddings

- One of the weaknesses of *one-hot representation* is that it treats each word as a thing unto itself, and it doesn't allow an algorithm to easily generalize the cross words (e.g. apple-orange, king-queen)
- Featured representation ⇒ Word embedding (e.g. gender, size, etc)
- Transfer learning & Work embeddings:
- 1) Learn word embeddings from large text corpus;
- 2) Transfer embedding to new task with smaller training set;
- 3) Continue to finetune the word embedding with new data
- Face encoding (any picture) vs. Word embedding (fixed vocabulary)
- Cosine similarity: $sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2}$
- Analogies example using word vectors:
- queen= $\text{argmax}_w \, sim(e_w, e_{king} - e_{man} + e_{woman})$
- Embedding matrix: $E \times o_j = e_j$, where o_j is the one-hot vector and e_j is the embedding vector ⇒ learn all the parameters in E
- Neural language model ⇒ feed the embedding vectors to FC & softmax

- Context/target pairs $\Rightarrow$ a few words right before the target word
- Skip-gram: $o_c \Rightarrow E \Rightarrow e_c \Rightarrow$ hierarchical softmax (to speed up) $\Rightarrow \hat{y}$
- Training set (content $\Rightarrow$ target): 1 positive case + k negative cases, where k = 5-20 for smaller datasets and k = 2-5 for larger datasets
- Negative sampling: $\mathbb{P}(y = 1|c, t) = \sigma(\theta_t^T e_c) \Rightarrow$ turn the giant softmax into binary classification with only $k + 1$ training examples
- Selecting negative examples: $\mathbb{P}(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_j^N f(w_i)}$, where f is frequency and N is the vocabulary size
- Global Vectors for word representation (GloVe):
- Define X_{ij} = #times j appears in context of i;
- Minimize $\sum_i^N \sum_j^N f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log(X_{ij}))^2$ where θ_i and e_j are symmetric, hence $e_w = \frac{e_w + \theta_w}{2}$;
- $f(X_{ij}) = 0$ if $X_{ij} = 0$ and $f(X_{ij})$ gives a meaningful amount of computation, even to the less frequent words
- Simple sentiment classification $\Rightarrow$ average all the embedding vectors $e_w = E \times o_w$ in the sentence and feed into softmax layer to predict $\hat{y}$
- RNN for sentiment classification $\Rightarrow$ take embedding vectors as input for the many-to-one RNN architecture and predict $\hat{y}$
- Word embeddings can reflect the gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model
- Addressing bias in word embedding:
- 1) Identify bias direction (e.g. $e_{he} - e_{she}$);
- 2) Neutralize $\Rightarrow$ for every word that is not definitional, project to get rid of bias (e.g. doctor, babysitter);
- Equalize pairs $\Rightarrow$ make sure the words that should be neutral are exactly same similarity (e.g. babysitter for grandfather & grandmother)
- A linear classifier can tell what words to pass through the neutralization step to project out this bias direction
- It is quite feasible to hand-pick most of the pairs to equalize

5.3 Sequence models & Attention mechanism

- Sequence to sequence model: encoder network + decoder network (e.g. French to English translation, image captioning)
- Machine translation $\Rightarrow$ conditional language model:
- $\mathbb{P}(y^{<1>}, ..., y^{<T_y>} \mid x^{<1>}, ..., x^{<T_x>}) = \mathbb{P}(y^{<1>}, ..., y^{<T_y>} \mid x)$
- $\hat{y} = \text{argmax}_{y^{<1>},...,y^{<T_y>}} \mathbb{P}(y^{<1>}, ..., y^{<T_y>} \mid x) \Rightarrow$ Beam search
- It's not always optimal for *greedy search* to pick best word one by one
- Beam search algorithm $\Rightarrow B =$ beam width:
- 1) Choose the Top B words which maximize $\mathbb{P}(y^{<1>}|x)$;
- 2) Choose the Top B pairs which maximize $\mathbb{P}(y^{<1>}, y^{<2>}|x)$, where $\mathbb{P}(y^{<1>}, y^{<2>}|x) = \mathbb{P}(y^{<1>}|x)\mathbb{P}(y^{<2>}|x, y^{<2>})$;
- 3) Continue for the next few words until termination by EOS symbol
- Beam search reduces to greedy search when $B = 1$
- Make the beam search a more numerically stable algorithm:
- $\text{argmax}_y \prod_{t=1}^{T_y} \mathbb{P}(y^{<t>} \mid x, y^{<1>}, ..., y^{<t-1>})$
- $\Rightarrow \text{argmax}_y \sum_{t=1}^{T_y} \log \mathbb{P}(y^{<t>} \mid x, y^{<1>}, ..., y^{<t-1>})$
- Length normalization $\Rightarrow$ reduce the penalty for outputting longer translations: $\text{argmax}_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log \mathbb{P}(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$, where $\alpha = 0$ means no normalization and $\alpha = 1$ means full normalization
- Larger $B \Rightarrow$ more possibilities, better result, compositionally slower
- Unlike BFS or DFS, Beam search runs faster but is not guaranteed to find exact maximum for $\text{argmax}_y \mathbb{P}(y \mid x)$
- Error analysis on Beam search: Human($\hat{y}$) vs. Algorithm(y^*)
- 1) $\mathbb{P}(y^* \mid x) > \mathbb{P}(\hat{y} \mid x) \Rightarrow$ Beam search is at fault;
- 2) $\mathbb{P}(y^* \mid x) \leq \mathbb{P}(\hat{y} \mid x) \Rightarrow$ RNN model is at fault;
- 3) Figure out what fraction of errors are due to Beam search vs. RNN

- Bleu (Bi-lingual evaluation understudy) Score:
- Uni-gram: $P_1 = \frac{\sum_{uni-grams \in \hat{y}} Count_{clip}(uni-grams)}{\sum_{uni-gram \in \hat{y}} Count(unigram)}$;
- N-gram: $P_n = \frac{\sum_{n-grams \in \hat{y}} Count_{clip}(n-grams)}{\sum_{n-grams) \in \hat{y}} Count(n-grams))}$;
- Combined Bleu score $= BP \times \exp\left(\frac{1}{N}\sum_{n=1}^{N} P_n\right)$, where BP stands for *brevity penalty* and MT represents machine translation:

$$BP = \begin{cases} 1, & \text{if MT length} > \text{Reference length} \\ \exp\left(1 - \frac{Reference\ length}{MT\ length}\right), & \text{otherwise} \end{cases}$$

- Attention model intuition: introduce the attention weights $\alpha^{<t,t^{'}>} \Rightarrow$ look at part of the sentence at a time rather than memorize the whole sentences and store it in the activations
- Attention model:
- Encoder: $x^{<t^{'}>} + a^{<t^{'}>} \Rightarrow c^{<t>}$; Decoder: $c^{<t>} + s^{<t>} \Rightarrow y^{<t>}$;
- $\sum_{t^{'}} \alpha^{<t,t^{'}>} = 1$; $c^{<t>} = \sum_{t^{'}} \alpha^{<t,t^{'}>} a^{<t^{'}>}$
- $\alpha^{<t,t^{'}>} = \frac{\exp(e^{<t,t^{'}>})}{\sum_{t^{'}=1}^{T_x} \exp(e^{<t,t^{'}>})} \Rightarrow$ amount of attention $y^{<t>}$ should pay to $a^{<t^{'}>}$, where $e^{<t,t^{'}>}$ and $\alpha^{<t,t^{'}>}$ are dependent on $s^{<t-1>}$ and $a^{<t^{'}>}$
- Speech recognition: audio clip $\Rightarrow$ transcript (phonemes $\Rightarrow$ DL)
- Connectionist temporal classification (CTC) cost $\Rightarrow$ collapse repeated characters not separated by blanks
- Trigger word detection $\Rightarrow$ set the target labels to be 1 when someone just finished saying the trigger word and remaining to be 0

■