Neural Network

Training of NNs

Activation functions

- $\sigma(x) = 1/(1+e^{-x}) -> (0,1), d\sigma = \sigma(1-\sigma) -> (0,1)$, gradient saturation
- tanh(x)→ (-1, 1), $dtanh(x) = 1 - tan^2h(x)$→(0,1)
- RELU→ max(0,x)
  - Leaky RELU→max(0.1x,x), ELU→max(e^x,x), etc

Back-Propagation

- forward pass: calculate f(x) from input, backwardness: calculate df(x) from output to input

Regularization

- **Bagging**: combine several models, construct a data set for each model, sampling with replacement from the original dataset, each dataset has the same size as the original one.It's beneficial if base classifiers have high variance
  - ESE= $E[(\frac{1}{k}\sum \epsilon_i)^2] = \frac{1}{k^2}E[\sum(e_i^2 + \sum_{j!=i}\epsilon_i\epsilon_j)] = \frac{1}{k}E[\epsilon_i^2] + \frac{k-1}{k}E[\epsilon_i\epsilon_j]$
  - if models are perfectly corrected then $E[\epsilon_i^2] = E[\epsilon_i\epsilon_j]$, ESE has no improvement; else $E[\epsilon_i\epsilon_j] = 0$
- **dropou**t: for each training example keep hidden units with probability, delete ingoing and outgoing links for eliminated hidden units
  - a vector input * a vector probability of 0 and 1. update input by 1/**p(to be learned)**
  - **Training** use ``inverted'' dropout
  - **Prediction** • Usually do not use dropout • no further scaling
  - Since each hidden unit can disappear with some probability, the network cannot rely on any particular units and have to spread out the weights
- **data augmentation**: rotation, flip,etc to create more data
- **early stopping: run the whole process, but take the optimal "stopping point" value**

Optimization

- stochastic gradient descent: slow convergence
- **SGD+ momentum**:α how quickly the contribution of previous gradients exponentially decay
  - g← $\frac{d}{d\theta}\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$
  - exponentially weighted averages of past gradients v← $\alpha v + (1-\alpha)g$
  - $\theta \leftarrow \theta - \epsilon v$
- **RMSProp(Root Mean Square):** global learning rate ϵ, decay rate ρ, initial parameter θ, δ small constant to stabilize devision by small numbers
  - g← $\frac{d}{d\theta}\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$

    - accumulated squared gradient r← $\rho r + (1-\rho) g \odot g$, discard history from extreme past
    - $\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$(element-wise operation), $\theta \leftarrow \theta + \Delta\theta$
- **Adaptive moment**s: step size ϵ, decay rate ρ, initial parameter θ, δ small constant to stabilize devision by small numbers
    - g← $\frac{d}{d\theta}\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$, t←t+1
    - biased first moment estimate s← $\rho_1 s + (1-\rho_1) g$, correct bias $\hat{s} = \frac{s}{1-\rho_1^t}$
    - biased second moment estimate r← $\rho_2 r + (1-\rho_2) g \odot g$, correct bias $\hat{r} = \frac{r}{1-\rho_2^t}$
    - $\Delta\theta \leftarrow -\frac{\epsilon \hat{S}}{\sqrt{\delta + \hat{r}}}$(element-wise operation), $\theta \leftarrow \theta + \Delta\theta$

CNN

- **feature**: $f(\sum wh)$, produced by activation function fed with a linear combination of a subset of units based on a **specific pattern of weights.** Same weight, move around the input under certain **stride→parameter sharing.**
- benefits:
    - reduce number of weights/parameters and over-fitting:
        - In DNN, an image with the size 200x200x3→ each hidden unit in layer 1 would have 200*200*3 = 120,000 weights. Consider 55*55*96 = 290,400 units in the first conv layer, and each has 11*11*3 = 363 weights and 1 bias. In total parameters=290400 units * 364 weights
        - In CNN, the input volume has size 32x32x3, if the filter size is 5x5, then each hidden unit in layer has 5x5x3 weights.
        - With parameter sharing, the first conv layer would now have only 96 unique set of weights (one for each depth slice). In total, 96*11*11*3 + 96(bias) = 34,944 parameters.
    - spatially local correlation
- Convo layer: filters, local in space but full along the entire depth. **Need to learn the best filter size.**
    - spatial size of output : 1+ (Input_size- filter_size+2_padding)/Stride
    - full size **= [1+ (W-F+2P)/S, 1+ (W-F+2P)/S, number of filters]**
    - input [5x5x3]+filter1 [3x3x3] + filter2[3x3x3] with stride=2→ output[3x3x2]
- Pooling layer: control over-fitting, reduce parameters and introduces no new parameter
    - accepts a volume of size W1xH1xD1, requires 2 hyper-parameters F and S, produces W2=(W1-F)/S+1, D2=D1
- deeper models are harder to optimize, increased depth, decreased performance due to gradient vanishing and exploding
- ResNet: Add an ``identity shortcut connection'' that skips one or more layers
    - stacks residual blocks, periodically skip conv layers. No FC layers

RNN

- examples
    - one(input image) to many(output tags)
    - many(input words) to one(pos/neg sentiment)
    - many to many: translation, video classification on frame level

- **parameter sharing:** $h^{(t)} = f(h^{(t-1)}, x^t) = g^t(x^t, x^{t-1}, ...)$
  - either the same f function for past hidden layer and present input, or a different g function given all input
  - **update function is the same for all hidden units**

training: input char→hidden layer→ output layer, choose the one with max probability

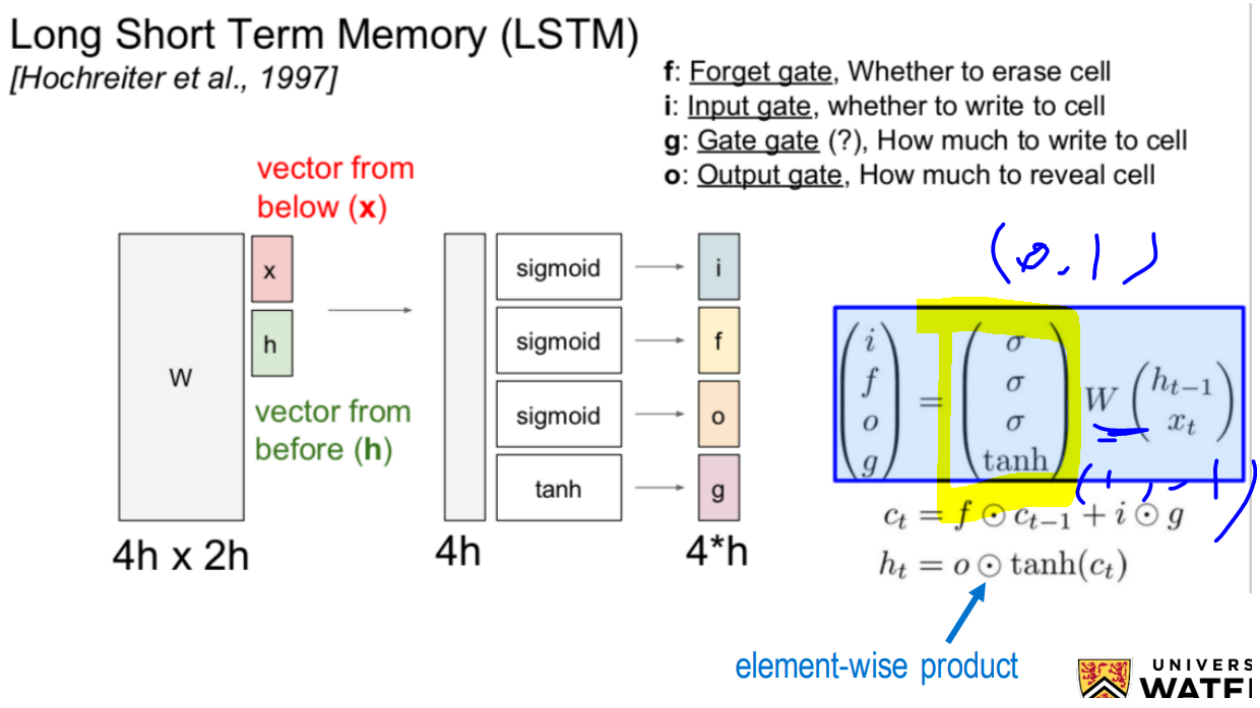
testing: initial input character→sample output, feed the result to next input

Vanilla RNN's gradient problem: $W = PDP^T$, to back-propagate we need to multiply W t times→ W^t

- if largest eigenvalue >1, **exploding gradient**
  - clipping gradient gradient← gradient * threshold / ||gradient||
- largest eigenvalue1 **vanishing gradient**
  - create paths along which the product of the gradients is near one

LSTM: back-propagation for c is element-wise multiplication(not matrix), and the value of f gate is different for each time step, better than multiplying W t times.

  - (input=0, forget=1): remember the previous value
  - (input=1, forget=1): add to the previous value(can act as counter)
  - (input=0, forget=0): erase the value
  - (input=1, forget=0): overwrite the value



GRU: Gated recurrent unit, simpler than LSTM

- R gate: current input and past hidden steps
  - $r_t = \sigma(W_{xr}x_t + W_{xr}h_{t-1} + b_r)$
- Z gate : similar to r, but different weight
  - $z_t = \sigma(W_{xz}x_t + W_{xz}h_{t-1} + b_z)$
- intermediate hidden state:
  - $h_{intermediate} = tanh(W_{xh}x_t + W_{hh}(r_t * h_{t-1}))$

- $h_t = z_t * h_{t-1} + (1 - z_t) * h_{intermediate}$

Bidirectional: input goes to both g and h, which act together for output
Deep RNN: $h_l^{(t)} = f(W_{l1}h_l^{(t-1)}, W_{l2}h_{l-1}^{(t)})$**, l hidden layer,t time step**

Generative Adversarial Networks

- objective of generative model: learn to represent a distribution(p_model), given the samples drawn from that distribution(p_data).Do not explicitly estimate a density function
  - $\theta = argmax_\theta E_{x\ p_{data}} log p_{model}(x|\theta)$**:** the best model
- Generator network: **minimize probability** of discriminator assigning the correct label
  - input: random noise
  - output: fake image
  - $E_{z\ p_z} log(1 - D(G(z))$, minimize objective such that D(G(z)) is close to 1
- Discriminator network: **maximize probability** of assigning the correct label, usually a DNN with parameters θ, such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
  - input: real and fake images
  - output: Real or Fake

$J = -(E_{x\ p_{data}} log D(x) + E_{z\ p_z} log(1 - D(G(z))$: real data and fake data generated by noise z.

- solution: $-(E_{x\ p_{data}} log D(x) + E_{z\ p_z} log(1 - D(G(z))$ => $-(E_{x\ p_{data}} log D(x) + E_{x\ p_{model}} log(1 - D(x))$
  - Fix G, D is optimal when the derivative is 0, $D(x) = p_{data}(X)/[p_{data}(X) + p_{model}(x)]$
  - plug back optimal D, objective is minimized when **p_model = p_data**
- saturating gradient: $log(1 - D(G(z)) \rightarrow -log D(G(z))$
  - in practice, use -log(D(G(z)) in generator network so that the quality of generated samples can improve faster.
- mode collapse: there exists a single fixed point that the generator thinks is the most optimal point to generate regardless of whatever input noise we feed it→causes low output diversity
  - sometimes our minmax problem behave like a maxmin problem.
    - fix D, minG = E log(1-D(x)). mode of D(x) x* becomes the most optimal point
  - solutions: minibatch feature
- tricks and tips for GAN
  - train with labels, instead of just images as input
  - one-sided label smoothing: reduce confidence in correct class. (real image→0.9)
  - batch normalization: allows us to use higher learning rates and less careful with initialization
- vector arithmetic: smilling women
- WGAN: Continuous and differentiable. can indicate quality. JS can't
- Pros: GANs can generate all entries of a sample in parallel, yielding greater generation speed. The design of the generator network has very few restrictions. Subjectively regarded as producing better samples than others
- Cons: Unstable training (non-convergence), Cannot solve inference queries such as p(x), p(z|x)

Auto encoder

- goal: copy its input to its output, extracts useful features. unsupervised learning
- **loss function:** x has k features
  - $L(x, \hat{x}) = -\sum_k x_k log\hat{x_k} + (1 - x_k)log(1 - \hat{x_k})$ for binary input
    - it makes sense because when X_k = 1, the optimal X_k* =1. when X_k=0, the optimal X_k*=0
  - $L(x, \hat{x}) = \sum_k (x_k - \hat{x_k})^2$
    - if we assume p(x|mean, variance)~N, and mean = W*h(x) and calculate negative log likelihood, we can derive this loss function
- **linear encoder and decoder**
  - input: each columns of X is a training instance, X=SVD
  - h(x)= $U_m^T x$, has less dimension than input. $\hat{x} = U_m h(x)$
  - If the features of input are mean-normalized(x-x_bar),encoder corresponds to principal component analysis (PCA)
- **under-complete auto-encoders**: dim h(x) < dim x
  - Good for the training distribution, Bad for other types of input
- **overcomplete autoencoders**: dim h(x) dim x, can just copy, no extract meaningful info
- **regularized**: $L(x, \hat{x}) + \lambda f()$
- **sparse**: constrain hidden units to be inactive most of the time
  - Sparsity parameter p, average activation of hidden unit j is $\hat{p} = \frac{1}{N}\sum_i^N h_j(x_i)$
  - Sparsity penalty $\sum_j^m plog\frac{p}{\hat{p}_j} + (1-p)log\frac{1-p}{1-\hat{p}_j}$ or $\sum_j^m KL(p||\hat{p}_j)$
    - since we want to make p=$\hat{p}$,and $KL(p||\hat{p}_j) = 0$ iff p=$\hat{p}_j$ (increases if $\hat{p}_j$ diverges
  - new Loss function $\sum_i^N L(x_i, \hat{x_i}) + \lambda \sum_j^m KL(p||\hat{p}_j)$
- **contractive autoencoder** $L(x, \hat{x}) + \lambda||\nabla_x h(x)||^2$**,** $||\nabla_x h(x)||^2 = \sum_j^m \sum_k^d (\frac{dh_j(x)}{dx_k})$
  - capture or ignore this variation→ encoder is resistant to small changes of input
- **denoising:** Randomly set input elements to 0 and add noise (e.g., Gaussian) to input, forcing the encoder to find correlation instead of copying
  - Loss function: compare output of decoder with noiseless input
- **VAE:** model the **underlying probability distribution of data** so that it could sample new data from that distribution (generative model)
  - Infer good values of z given data x by calculating posterior p(z|x), but p(z|x) is hard to calculate, thus approximate p(z|x) by a family of distributions $q_\lambda(z|x)$ (if q were Gaussian, λ denotes mean and variance)
    - objective: minimize KL divergence between q(z|x) and p(z|x)
    - loss function = $-E_{z\sim q_\theta(z|x_i)}[logp(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$
    - First term (negative log likelihood): encourages decoder to reconstruct data
    - Second term: minimize distance between q(z|x) and p(z), p(z): N (0,1)
    - Output of the incouder is lists of μ, σ, input to decoder is a sample from $N(\mu, \sigma^2)$ at random
    - to generate new samples, sample z ~ N(0,1) and feed it into the decoder network

Decision Tree

- $H(X) = -E_X log_b p(X) = -\sum_i^n p(x_i) log_b p(x_i)$**,** larger entropy larger uncertainty
- **ID3**: pick the maximum i**nformation gain(S,A) =** $H(S) - \sum_{V\in Values(A)} \frac{|S_V|}{|S|} H(S_v)$, the difference of entropy before split and the **weighted** entropy after split.
  - Cannot ensure an optimal solution, non-robust, small change in the training data can result in a big change in the tree

    - Overfitting. If there is no label noise, training set error is always 0
- **Continous**: sort values of A in ascending order then pick the mid points as t, best t $t* = argmin_{t \in T} \frac{|S_{A<t}|}{|S|} H(S_{A<t}) + \frac{|S_{A \geq t}|}{|S|} H(S_{A \geq t})$
- **Gain Ratio** = Gain(S,A)/SplitInfo(S,A), SplitInfo(S,A)=$- \sum_{V \in Values(A)} \frac{|S_V|}{|S|} log \frac{|S_V|}{|S|}$

Ensemble methods

- assuming independent data and independent hypothesis, as the number of classifiers approaches infinity, the probability of correct prediction -> 1
- **committees:** 20 models for 20 independent training datasets, average 20 models (close to true function)
- **AdaBoost**: a **weighted** form **w** of the data set, depend on performance of previous classifiers where misclassified points are given greater weight
    - calculate the weight for models $\alpha_m = ln(\frac{1-\epsilon_m}{\epsilon})$
    - calculate the weight of data set $w_n^{m+1} = w_n^m exp\{\alpha_m I(y_m(x_n)! = t_n)\}$
    - Final prediction: **weighted** performance **a** of base classifiers $sign(\sum_m \alpha_m y_m)$
        - exponential error function: $E = \sum_n^N exp\{-t_n f_m(x_n)\}, f_m(x) = \frac{1}{2} \sum_l^m \alpha_l y_l(x)$ goal is to minimize E wrt a_l(weight) and the parameters of y_l(x)(hypotheis)
            - t_n: target (output) of the n-th data point
            - f_m($\boldsymbol{x}$_n): a classifier that is a linear combination of base classifiers
- **tree bagging:** Bootstrap data sets (add variability in dataset)→fit trees to these data sets→ Prediction: average/majority vote
- **random forest:** select a random subset of attributes (e.g., m attributes) at each candidate split
    - m is a hyperparameter, classification: m=# of attributes, regression: m= # of attributes/3
- increasing correlation increases forest error rate, increasing strength of individual

trees decreases forest error rate