1. Lecture 1: Logistic Regression

Idea: pred $y = P(y=1|x)$
- Simple: $\hat{y} = wx + b$
- If x is vector: $\hat{y} = w^T x + b$
- sigmoid out $0 \le \hat{y} \le 1$ (i.e., prob):
- $\sigma(z) = \frac{1}{1+e^{-z}} \to \hat{y} = \sigma(w^T x + b) = \hat{y}(z)$

1.1. Loss and Cost Functions

Loss function is $\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$, but it's not convex ⇒ problematic for gradient descent, so use:

$\mathcal{L} = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$
- For $y = 1 \Rightarrow \mathcal{L} = -\log(\hat{y})$, want $\hat{y}$ close to 1.
- For $y = 0 \Rightarrow \mathcal{L} = \log(1-\hat{y})$, want $\hat{y}$ close to 0

Derivatives: $\frac{d\mathcal{L}(a,y)}{dz} = a - y$

$$\frac{d\mathcal{L}(\hat{y},y)}{d\hat{y}} \equiv \frac{d\mathcal{L}(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

Cost function eval performance on entire train set:

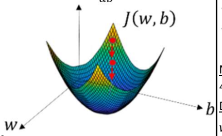
$J(w,b) = -\frac{1}{m}\sum_{i=1}^m \mathcal{L}(y,\hat{y}) = -\frac{1}{m}\cdot \sum_{i=1}^m y^i \log \hat{y}^i + (1-y^i)\log(1-\hat{y}^i)$

Vectorized form: $J(w,b) = -\frac{1}{m}\cdot [Y^T \log(A) + (1-Y^T)\log(1-A)]$

1.2. Gradient Descent

Idea: adapt params to min cost.
Alg: iteratively apply

$$w = w - \alpha \frac{dJ(w)}{dw} = w - \alpha dw$$
$$b = b - \alpha \frac{dJ(b)}{db} = b - \alpha db$$



1.3. Logistic Regression Vectorized Alg (m samples and n features)

$J = 0; db = 0;$ # Init
Init vector of zeros:
$dw = np.zeros(n,1);$
For $i = 1$ to m:
  $z^i = w^T x^i + b$ # Forward prop
  $a^i = \sigma(z^i)$ # Calc activation
  $J += -[y^i \log a^i + (1-y^i)\log(1-a^i)]$ # Calc Loss
  $dz^i = a^i - y^i$ # Calc derivs
  $dw += x^i dz^i$ # Vector operation
  $db += dz^i$
$J /= m; db /= m$ # Avg
$dw /= m;$ # Vector operation
- Note: vec form ($dw += x^i dz^i$) faster than using for loop.

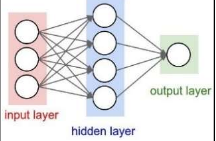
1.4. Vectorizing LogReg Grad Des

Input:
- $X = \begin{pmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{pmatrix}$: is $n \times m$ (n features, m samples) where each column is sample
- $w = [w_1 \dots w_{n_x}]$: is $1 \times n$
- $Y = [y^1 \dots y^m]$: are labels

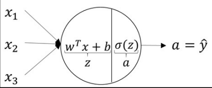
Algorithm:
Forward propagation:
$Z = [z^1 \dots z^m] = w^T X + [b_1 \dots b_m] = w^T X + b$
Calc matrix of all activations:
$A = [a^1 \dots a^m] = \sigma(Z)$
Calc loss with matrix operations:
$J = -\frac{1}{m}\cdot [Y^T \log(A) + (1-Y^T)\log(1-A)]$
Calculate the derivative:
$dz = A - Y = [a_1 - y_1, \dots, a_m - y_m]$
Calculate the gradient:
$dw = \begin{pmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{pmatrix} \begin{bmatrix} dz^1 \\ \vdots \\ dz^m \end{bmatrix} = \frac{1}{m} X dZ^T; db = \frac{1}{m}\sum_{i=0}^m dz^i$
Calc Gradient Descent
$w = w - \alpha dw;\ b = \alpha db$

2. Feedforward Neural Networks FNN

$a^{[0]} = X_{3\times1}$ $w^{[1]}_{4\times3}$ $w^{[2]}_{1\times4}$
$b^{[1]}_{4\times1}$ $b^{[2]}_{1\times1}$
$a^{[1]}_{4\times1}$ $a^{[2]}_{1\times1} = \hat{y}$



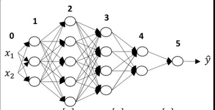
The perceptron:



2.1. Grad Descent for NNs

Input: x, Layer i params: $W^{[i]}, b^{[i]}$
Forward propagation:
$Z^{[1]} = W^{[1]}X + b^{[1]} \to$
$A^{[1]} = \sigma(Z^{[1]}) \to$ # This is X now
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \to$
$A^{[2]} = \sigma(Z^{[2]}) \to$
$\mathcal{L}(a^{[2]}, y)$ # Calculate Loss
Backward propagation:
$dz^{[2]} = a^{[2]} - y \to$
$dW^{[2]} = dz^{[2]} a^{[1]^T} \to$
$db^{[2]} = dz^{[2]} \to$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]}) \to$
$dW^{[1]} = dz^{[1]} x^T; db^{[1]} = dz^{[1]}$

2.2. Deep Neural Networks (DNN)



Neurons: $n^{[1]} = 3, n^{[2]} = 5, n^{[3]} = 4, n^{[4]} = 2, n^{[5]} = 1$

Dimensions: $Z^{[l]}_{n^{[l]}\times m} = w^{[l]}_{n^{[l]}\times n^{[l-1]}} X_{n^{[l-1]}\times m} + b^{[l]}_{n^{[l]}\times m}$

2.3. Forward and Backward Prop

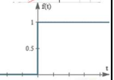
$X \to ReLU \to a^{[1]} \to ReLU \to a^{[2]} \to Sigmoid \to \hat{y} \to \mathcal{L}(\hat{y}, y) \to da^{[3]} = -\frac{y}{a} + \frac{1-y}{1-a} \to$

$\begin{cases} da^{[2]} \to \\ dw^{[3]} \\ db^{[3]} \end{cases} \begin{cases} da^{[1]} \to \\ dw^{[2]} \\ db^{[2]} \end{cases} \begin{cases} dw^{[1]} \\ db^{[1]} \end{cases}$

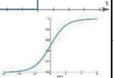
2.4. Activation Functions

Identity:
$f(x) = x; f'(x) = 1$

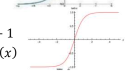
Step:
$f(x) = \begin{cases} 0, x < 0 \\ 1, x \ge 0 \end{cases}$
$f'(x) = \begin{cases} 0, x \ne 0 \\ \infty, x = 0 \end{cases}$



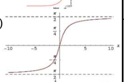
Sigmoid:
$f(x) = \frac{1}{1+e^{-x}}$
$f'(x) = f(x)(1 - f(x))$



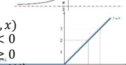
TanH:
$f(x) = \frac{2}{1+e^{-2x}} - 1$
$f'(x) = 1 - f^2(x)$



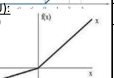
Arctan:
$f(x) = \tan^{-1}(x)$
$f'(x) = \frac{1}{x^2+1}$



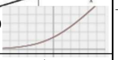
ReLU:
$f(x) = \max(0, x)$
$f'(x) = \begin{cases} 0, x < 0 \\ 1, x \ge 0 \end{cases}$



Leaky ReLU (PReLU):
$f(x) = \begin{cases} \alpha x, x < 0 \\ x, x \ge 0 \end{cases}$
$f'(x) = \begin{cases} \alpha, x < 0 \\ x, x \ge 0 \end{cases}$



SoftPlus:
$f(x) = \ln(1 + e^x)$
$f'(x) = \frac{1}{1+e^{-x}}$



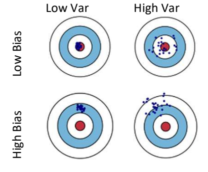
Universal Approximation Theorem: 2 layers are enough to approximate any continuous function for inputs in specific range.

3. Lecture 2: Regularization & Parameter Optimization

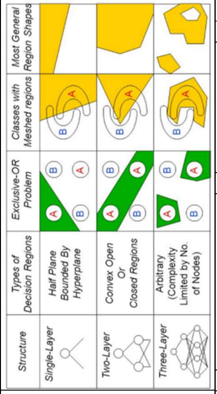
Bias: diff between model expected (or avg) pred and real value:

$$Bias\left(\hat{f}(x)\right) = E\left(\hat{f}(x) - f(x)\right)$$

Variance: model pred variability



- High bias ⇒ underfitting
- High Var ⇒ overfitting



3.1. Regularization

Apply constraints on learning alg s.t. generalizes better (not overfit). Constraints on estimators ⇒ trading high bias for low variance.

3.2. Parameter Norm Penalties

$$\hat{J}(w,b) = J(w,b) + \varepsilon \Omega(w)$$

- Reg only applied to w (biases are easier to config, need less data)
- Apply diff α for diff layers can be optimal but very expensive to find ⇒ use 1 value for all layers

3.3. L2 Norm Regularization

Idea: L2-reg (also weight decay, ridge regression) assumes model with small weights is simpler than large weights ⇒ penalizing large weights makes weights smaller.
- Leads to smoother model in which output changes more slowly as input changes.

$\Omega = \frac{1}{2}\|w\|_2^2$
$\hat{J}(w,X,y) = \frac{\varepsilon}{2} w^T w + J(w,X,y)$

Grad for updated cost:
$\nabla_w \hat{J}(w,X,y) = \varepsilon w + \nabla_w J(w,X,y)$

Update step for w:
$w = w - \alpha(\varepsilon w + \nabla_w J(w,X,y)) = (1 - \alpha\varepsilon)w - \alpha \nabla_w J(w,X,y)$

3.4. L1 Norm Regularization

L1 stay relatively const & doesn't scale lin (unlike L2). L1 is sparser ⇒ used in feature select.

$\Omega = \|w\|_1 = \sum_i |w_i|$
$\hat{J}(w,X,y) = \varepsilon\|w\|_1 + J(w,X,y)$
$\nabla_w \hat{J}(w,X,y) = \varepsilon sign(w) + \nabla_w J(w,X,y)$

3.5. Adding Noise to Data

Idea: add random (Gaussian) noise, to "nudge" model to regions where it is relatively insensitive to noise.
- For small noise ψ and adding ψ to each sample ($\hat{y}(x) \to \hat{y}_\psi(x)$), ⇒ min cost $J_\psi(x,y) = (\hat{y}_\psi(x) - y)^2$ ⇔ min cost with reg param.

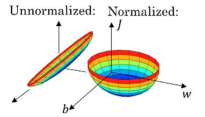
3.6. Adding Noise to the Output

Problem: SoftMax layer out ∈ [0,1], thus binary clf can only out approximations of 0s & 1s ⇒ may make net increase weights to get closer to 0,1

Possible solution: instead of using original labels "as is" (ie, $1, \dots, k$), set prob to be $[\frac{1}{k}\varepsilon, 1 - \left(1 - \frac{1}{k}\right)\varepsilon]$, for small const ε and # of classes k.

3.7. Input Normalization

1. Subtracting mean: $x^i = x^i - \mu = x^i - \frac{1}{m}\sum_{i=1}^m x^i$
2. Norm var: $x^i = x^i[\sigma^2]^{-1} = x^i\left[\frac{1}{m}\sum_{i=1}^m (x^i)^2\right]^{-1}$



3.8. Early Stopping

Problem 1: overfitting train.
Solution: use val, stop train when performance on val not improving.
Problem 2: val reduces train size
Solution: train with val, to decide when to stop, then train again with all data for same # of epochs.
- Low-cost, effective form of reg.

3.9. Dropout

Idea: can't rely on any 1 feature, so force use multiple features. Also, it causes weights to decrease, like L2
prob to keep node in each layer:
$keep\_prob$
$d_i = np.rand(a_i.shape[0], a_i.shape[1]) < keep\_prob$
$a_i = a_i \cdot d_i$ # cover part of layer
Correct a_i for miss nodes
$a_i /= keep\_prob$
- No dropout for test set

3.10. Adversarial Training

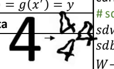
Idea: gen samples x' that while close to x yield very diff model out.
- This weakness is since NNs are lin, despite their non-lin activatio
- Change of ε to input can change output by as much as $\varepsilon\|w\|_1$.

Ad. samples makes model more const around input x instead of lin. ⇒ Model is robust to small changes

$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{x}, y) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, y) + (1-\alpha)J(\boldsymbol{\theta}, \boldsymbol{x} + \varepsilon\,\mathrm{sign}\,\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y)$ for $\alpha J(\boldsymbol{\theta}, \boldsymbol{x}, y)$ train target, rest ad-reg

Semi-supervised usage:
$x \in X$ is sample s.t. $g(x) = y$.
Find $x' \approx x$ s.t. $g(x') = \hat{y} \ne y$.
⇒ Model tries to $g(x) = g(x') = y$

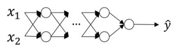
3.11. Augmenting Data



3.12. Mini-Batch Gradient Descent

Idea: split data to batches for efficient process of large dataset.
Alg: batches $x^{\{1\}} \dots x^{\{n\}} \in X$ size m
$For\ t = 1\ to\ n$: # for each batch
  $For\ i = 1\ to\ L$: # Forward prop
    $Z^{[i]} = W^{[i]T}A^{[i-1]} + b^{[i]}$
    $A^{[i]} = g^{[i]}Z^{[i]}$
  $J^{\{t\}} = \frac{1}{|m|}\sum_1^m \mathcal{L}(y^{\{t\}}, \hat{y}^{\{t\}}) + \frac{1}{2m}\|w^{\{t\}}\|_2^2$
  $For\ i = L\ to\ 1$: # Backward prop
    $dW^{[i]} = dZ^{[i]}A^{[i-1]T}$
    $db^{[i]} = dZ^{[i]}$
- Note: it's a compromise since grad on 1 batch isn't best globally
- Increase params value variation

3.13. Vanishing & Exploding Grad



Problem: for $g(z) = z$
⇒ $z^i = w^{iT}a^{i-1} \Rightarrow \hat{y} = \prod_i^1 w^i x$.
- Exploding weights:
  $\forall i\ w^i = 1.5 \Rightarrow \hat{y} = 1.5^n x$
- Vanishing weights:
  $\forall i\ w^i = 0.5 \Rightarrow \hat{y} = 0.5^n x$
- Same problem for gradients

(Partial) Solution: smart param init.

3.13.1. Param Init Strategies

- Decides if model converges and converges time.
- Init must break symmetry (init all to 0 is bad).
- Generally, weights should be around 1 (not too big or small).

3.13.2. Random distribution

Idea: (most common) init weights from random Gaus/uniform dist. Set biases to heuristically chosen consts. Larger weights help break symmetry faster ⇒ converge faster (but can cause exploding grads).

Best practices:
- m inputs layer: $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$.
- m inputs and n outputs layer: $U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}})$ (Glorot, 2010)

3.13.3. Activation Specific Init

For n, m # of layer inputs, outputs:

ReLU: $W^i = np.randn(shape)\cdot\sqrt{\frac{2}{n}}$

For $var(W^i) = \frac{2}{n}$ (better than $\frac{1}{n}$)

Tanh:
1. Xavier init: $W^i = np.randn(shape)\cdot\sqrt{\frac{1}{n}}$
2. $W^i = np.randn(shape)\cdot\sqrt{\frac{2}{n+m}}$

3.14. Adjusting Learning Rate (LR)

Problem: grads vary from batch to batch ⇒ may cause params to oscillate and longer converges.
Solution: reduce LR over time
1. By epochs: $\alpha = \frac{\alpha_0}{1 + decayRat\ \cdot epoc}$
2. Exp. decay: $\alpha = 0.95^{epoc} \cdot \alpha_0$
3. Poly. decay: $\alpha = \frac{k}{\sqrt{epoch}}\cdot\alpha_0$
4. Lin. decay: reduce α by const every fix # of epochs.

4. Grad Des Optimize. Techniques

4.1. Momentum

Idea: calc exp weighted avgs for grads and update weights with new values for faster convergence.
Alg: For each iteration, calc dW, db on current batch, then

$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$
$v_{db} = \beta_1 v_{db} + (1-\beta_1)db$
$W = W - \alpha v_{dW}, b = b - \alpha v_{db}$
- Common choice $\beta_1 = 0.9$

4.2. RMSprop (Root Mean Square)

Idea: apply dampening effect to each individual element ⇒ varying elements will moderate more.
Alg: For each iteration, calc dW, db on current batch, then:
squaring is elementwise:
$sdw = \beta_2 sdw + (1-\beta_2)dw^2$
$sdb = \beta_2 sdb + (1-\beta_2)db^2$
$W -= \alpha\frac{dw}{\sqrt{sdw+\varepsilon}}, b -= \alpha\frac{db}{\sqrt{sdb+\varepsilon}}$
- $\varepsilon = 10^{-8}$ is added for non-zero s
- Common choice $\beta_2 = 0.999$

4.3. Adam (Adaptive Moment Est)

Idea: RMSprop + momentum.
Alg: For each iteration, calc dW, db on current batch, then:
Calc v_{dW}, v_{db} # Use momentum
Calc sdw, sdb # Use RMSprop
Fix bias:
$v_{dW} /= (1-\beta_1), v_{db} /= (1-\beta_1)$
$sdw /= (1-\beta_2), sdb /= (1-\beta_2)$
$W -= \alpha\frac{v_{dW}}{\sqrt{sdw+\varepsilon}}, b -= \alpha\frac{v_{db}}{\sqrt{sdb+\varepsilon}}$

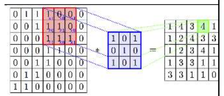
4.4. Batch Normalization

Alg: for $z^1 .. z^m$ values of layer i:
calc mean and var:
$\mu = \frac{1}{m}\sum_i z^i, \sigma^2 = \frac{1}{m}\sum_i (z^i - \mu)^2$
$z^i_{norm} = \frac{z^i - \mu}{\sqrt{\sigma^w + \varepsilon}}$ # Norm each z^i
Use learning params to avoid forcing z^i to have std mean & var:
$\tilde{z}^i = \gamma z^i_{norm} + \beta$
- ε is small const for non-zero div.
- If $\gamma = \sqrt{\sigma^2 + \varepsilon}, \beta = \mu \Rightarrow \tilde{z}^i = z^i$

Pros:
- More robust for data changes (covariate shift).
  - layer outputs norm, limits dist of values that will be seen by subsequent layer
  - Effective in DNN, wer changes snowball to deep layer
  - Layers learn more independent.
- Also performs slight reg when applied on batches
  - Mean, var change constantly since mean and var are computed for each batch.
  - Introduces some noise to data, which does regularization.

5. Lecture 3: CNNs

5.1. Convolution and Filters



$n \times n \to (n - f + 1) \times (n - f + 1)$

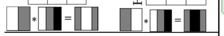
Edge detection:

Vertical

1	0	-1
1	0	-1
1	0	-1

Horizontal

1	1	1
0	0	0
-1	-1	-1



5.1.1. Padding

Problem 1: filter shrinks img dims.
Problem 2: less filters on corners ⇒ smaller impact of corners.
Solution: pad img with 0s around.
size of conv output with padding:
$\boldsymbol{n + 2p - f + 1}$. Eg, 8×8 image after 3×3 filter $\to (8-3+1)\times(8-3+1) = 6\times6$ img $\to 8\times8$ img after pad
- Valid conv: conv without padding
- Same conv: pad s.t. output after filter is same as input. ie,
  $n + 2p - f + 1 = n \to p = \frac{f-1}{2}$

5.1.2. Strided conv

Slide s pixels after each conv (instead of 1). New dims:
$\left\lfloor\frac{n+2p-f}{s} + 1\right\rfloor \times \left\lfloor\frac{n+2p-f}{s} + 1\right\rfloor$

5.1.3. RGB conv

Apply filter with channels (always equal # to input channels) $f\times f\times3$ on $n\times n\times3$ RGB input to get $(n-f+1)\times(n-f+1)$ image. E.g., $6\times6\times3 * 3\times3\times3 = 4\times4$

5.1.4. Applying Multiple Filters

Apply diff filters and stack them. E.g., apply filters that only look for features in specific channel.

5.2. Conv Layer

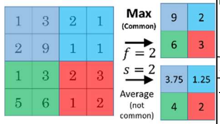
Idea: use conv where filters weights are the params
$6\times6\times3 * 3\times3\times3 = ReLU(4\times4 + b), params\# = f\cdot f\cdot c + 1$
⇒conv layer has fewer params. E.g., 10 filters of $3\times3\times3$ have $3\cdot3\cdot3 = 27$ and 1 bias ⇒ $28\cdot10 = 280$
General thumb rule in CNNs: deeper you go → smaller height and width → but higher # of channels

5.2.1. 1X1 Conv Layer

Idea: reduce layer dims. Acts as a FC layer for small part of prev layer.
More efficient than FC. E.g.:
- $6\times6\times32 * 1\times1\times32 = 6\times6$
- For 32 filters:
  $28\times28\times192 * [1\times1\times192]^{[32]} = 28\times28\times32$

5.3. Pooling layers

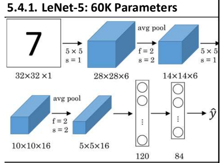
Idea: keep most important features in each area to pass forward intact.



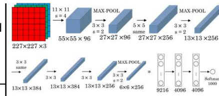
- Don't have params to update.

5.4. CNNs Architectures

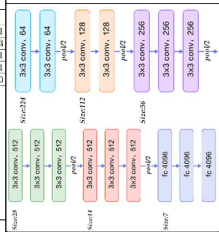
5.4.1. LeNet-5: 60K Parameters



5.4.2. AlexNet: 60M Parameters

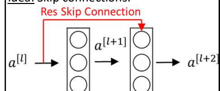


5.4.3. VGG: 138M Parameters



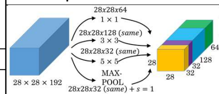
5.4.4. Residual Networks (ResNet)

Problem: vanishing/exploding grads
Solution: ResNet stop improving instead of decrease performance
Idea: Skip connections.

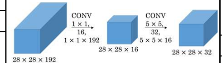


Regular FFN (black) vs ResNet (red):
$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$
$a^{[l+1]} = g(z^{[l+1]})$
$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$
$\cancel{a^{[l+2]} = g(z^{[l+2]})}$
$\boldsymbol{a^{[l+2]} = g(z^{[l+2]} + a^{[l]})} = g(w^{[l+1]}a^{[l+1]} + b^{[l]} + a^{[l]})$
- If layer isn't useful, L2 brings its params close to 0, thus $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$ (for ReLU).
- Grads are evenly split and summed in layer 1

5.4.5. Inception Network

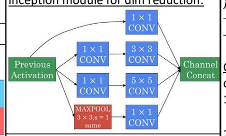


Idea: let net decide what layer and filters to use (any of the 4 options)
Problem: high computational cost. For the 5×5 conv we have 32 filters each of dim $5\times5\times192 \Rightarrow 28\cdot28\cdot32\cdot5\cdot5\cdot192 \approx 120M$
Solution: use $1\times1\times16$ conv before 5×5 conv.



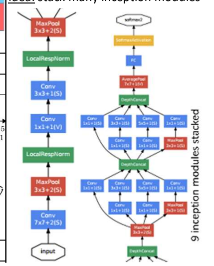
1. $28\cdot28\cdot16\cdot192 = 2.4M$
2. $5\cdot5\cdot16\cdot28\cdot28\cdot32 = 10M$
⇒ $2.4M + 10M \ll 120M$

Inception module for dim reduction:



5.4.6. GoogLeNet

Idea: stack many inception modules



5.5. Image Classification vs Localization & Detection

Idea: $\{P_c, b_x, b_y, b_h, b_w, C_1, C_2, C_3\}$ net output, where P_c is prob of some obj in img, b_x, b_y, b_h, b_w are bounding box coord, $C_1, \dots, C_k$ are prob of obj class C_i appear in img.

$$\begin{cases} \mathcal{L} = \frac{1}{2}\sum(\hat{y}_i - y_i)^2 & if\ P_c = 1 \\ \mathcal{L} = \frac{1}{2}(\hat{y}_1 - y_1)^2 & if\ P_c = 0 \end{cases}$$

5.5.1. Sliding Windows Detection

Idea: slide window over image and predict if obj in window,
Problem: inefficient.
Solution: YOLO algorithm

5.5.2. YOLO Algorithm

For each cell ∈ image grid:
If obj center ∈ cell, mark as contain
- Output for each cell is of form $\{P_c, b_x, b_y, b_h, b_w, C_1, C_2, C_3\}$, for b_x, b_y bound box center.
- Net output is $H\times W\times8$, for H, W are the grid dims.
- Note: boxes sizes can exceed greed boundaries

5.5.2.1.Intersect over Union (IoU)

Idea: measure overlap between two bound boxes:

$$IoU = \frac{\cap\, box_i}{\cup\, box_i}$$

If $IoU \ \ge\ 0.5 \Rightarrow$ box is correct

YOLO Problem: several bound boxes may contain obj (i.e., multi detections of same object).
Solution: Non-max Suppression alg.

5.5.2.2.Non-Max Suppression Alg

For each target class:
  $Y = [P_c, b_x, b_y, b_h, b_w]$
  If $P_c < 0.6$ discard
  While exists remaining boxes:
    $b =$box with max P_c
    Discard boxes with $IoU > 0.5$

YOLO Problem: grid can only detect one obj, not multiple objs.
Solution: Anchor boxes alg.

5.5.2.3.Anchor Boxes Algorithm

Idea: output concatenated anchor boxes instead of one, then in train use the anchor with highest IoU. I.e., instead of $k\times k\times(k+5)$ for k classes, output $k\times k\times[\#anchors\cdot(k+5)]$

5.6. Siamese Net Face Recognition

Idea: 2 identical CNNs which encode input images x_1, x_2 to $f(x_1), f(x_2)$
Loss: $d(x_1, x_2) = \|f(x_1) - f(x_2)\|_2^2$
For d is imgs similarity (deg of diff)
- Small $d \Leftrightarrow x_1, x_2$ are same
- Adding new person is 1 more comparison ⇒ inexpensive

5.6.1. Triplet Loss

For img A anchor P pos N neg, need pos dist $d(A,P) \le$ neg dist $d(A,N)$.
↳ $d(A,P) - d(A,N) + \alpha \le 0$
⇒ $\mathcal{L}(A,P,N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$
- α (margin) added for non-zero
- For at least some persons more than 1 img is required (else no P)

Choosing triplets A, P, N: Random choice too easy ⇒ NN don't learn.
1. Choose "semi-hard" examples within margin α (not too hard)
2. Put pos in 1 batch, assign rand negs (in margin), select triplets

5.6.2. Face Verification & Bin Clf

Alternative Triplet Loss: apply same net twice, plug outputs to LogReg, which outputs loss: $\hat{y} = \sigma(\sum_1^{128} w_i \left|f\left(x_k^{(i)}\right) - f\left(x_k^{(j)}\right)\right| + b)$

5.7. Visualize DNN

Idea: deeper layers learn more complex representations since they get larger images.
Alg: pick unit in layer 1, find 9 img patches that max its activation, repeat for other units & layers.
⇒ Layer 1 looks for simple things (edges/textures) since it sees small part of img while deeper layers get larger imgs & learn complex reps.

5.8. Neural Style Transfer

Idea: with images C content, S style, gen img $G = C$ with style S. Done using shallow & deep layers features of train CNN (transf learn).
Cost: $J(G) = \alpha\, J_{content}(C,G) + \beta\, J_{style}(S,G)$ for J is images similarity.
Alg: rand init G: $100\times100\times3$.
$\min J(G)$ with grad des ($G -= dG$).

5.8.1. Content Cost Function

$J_{content}(C,G) = \frac{1}{2}\left\|a^{[l][G]} - a^{[l][G]}\right\|^2$ for l layer l of a trained CNN (e.g., VGG), $a^{[l](C)}, a^{[l](G)}$ are activation of layer l on images. If $a^{[l](C)}, a^{[l](G)}$ are similar, then they have same content.
- Choose middle layer l since low layers create pix2pix reconstruct.

5.8.2. Style Cost Function

Idea: image style is the correlation between activeations across channels.
Style matrix (Gram matrix):

$$G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}}\sum_{j=1}^{n_W^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'}$$

For H, W, C are height, width, channels
$a^{[l]}_{i,j,k}$ = activation at (H, W, C)
- $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

Calc $G^{[l][S]}_{kk'}, G^{[l][G]}_{kk'}$ for style & gen.
- Corr activations make $G^{[l]}_{kk'}$ large (unnormalized cross covariance since mean not subtracted).

Style cost function:
- For 1 layer: $J^{[l]}_{style}(S,G) = \left(2n_H^{[l]}n_W^{[l]}n_C^{[l]}\right)^{-2}\sum_k \sum_{k'}\left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)$
- For entire architecture: $J_{style}(S,G) = \sum_l \lambda^l J^{[l]}_{style}(S,G)$

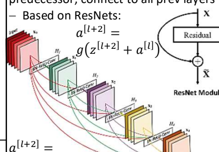
5.9. Enhancing CNN Architecture

Problem: deeper networks give better performance, but suffer from (1) vanishing grad risk, and (2) more run time & params.
Solutions:
1. Squeezing (ie architectures/layers/blocks that reduce channels dims at specific locations)
2. More interconnected architectures that better shares info forward (& grads backwards)

5.9.1. Densely Connected Networks

Idea: instead of connecting layer with predecessor, connect to all prev layers
- Based on ResNets:

$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$



$a^{[l+2]} = g(z^{[l+2]}, a^{[0]}, \dots, a^{[l+1]})$

Problem: multiple filter concats ⇒ hard to handle net **dim**.
Solution: use (relatively) small # of filters (eg 12) per layer (high layer connectivity more than compensates).
After each layer use BatchNorm + ReLU + 3x3 conv. Create **dense blocks**, each applying this approach, connected by **transition layers**.
Transition layers: consists of BatchNorm, 1x1 conv, 2x2 pool layer
- To improve efficiency: reduce # of channels passed forward layers. For m channels, transition can only transfer $0 < \theta < m$ channels.

5.9.2. Squeeze & Excitation Nets

Problem: each filter learns locally, thus can't utilize context info out of its region.
Solution: enable net to extract global info, then selectively provide to various layers (like gating/attention)

5.9.2.1.Squeeze

Idea: squeeze global spatial info to narrow channel. Obtain channel-wise stats using global average (out is like collection of all local descriptors).

$$z_c = F_{sq}(u_c) = \frac{1}{H\times W}\sum_{i=1}^H\sum_{j=1}^W u_c(i,j)$$

5.9.2.2.Excitation

Problem: hard to capture inter-channel dependencies (which is hard).
Need meet two criteria:
1. Flexibility: learn non-lin interactions.
2. Non-mutually-exclusive: makes sure that multiple dependencies can be modeled for channel

Solution: sigmoid.
- For efficiency (ie reduce # of params), wrap sigmoid with 2 FC layers (before and after)

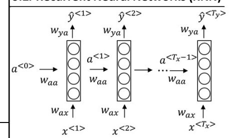
6. Lec 4: Recurrent Neural Networks

6.1. Representing Words

One-hot dot (sparse) representation of words: with vocab vec V, word $x^{<i>}$ is vec of 0s (size V) and single 1 matching word pos in V.
- For $x \notin V$ enc as "unknown".
- $x^{(i)<t>}, y^{(i)<t>}$: input, out of term t of sample i
- T_x: # of terms in sample x

Problem using with NN:
1. Diff ins, outs lens in diff exampls.
2. Don't share features learned across diff text pos.

6.2. Recurrent Neural Networks (RNN)



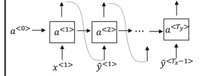
- $a^{<t>} = g(w_{aa}a^{<t-1>} + w_{ax}x^{<t>} + b_a)$ – tanh/ReLU
- $\hat{y}^{<t>} = g(w_{ya}a^{<t>} + b_y)$ – sigmoid/SoftMax
- If bin clf ⇒ Sigmoid, else SoftMax.
- Notation: $W_a[a^{<t-1>}, x^{<t>}] + b_a \equiv w_{aa}a^{<t-1>} + w_{ax}x^{<t>} + b_a$

Forward Prop Cost of single sample:
$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = \sum_{i=1}^t \mathcal{L}^{<i>}(\hat{y}^{<i>}, t^{<i>}) = \sum_{i=1}^t[-y^{<t>}\log\hat{y}^{<t>} - (1 - y^{<t>})\log(1 - \hat{y}^{<t>})]$
- For m batches calc average.

Backprop Through Time (BPTT): With loss $\mathcal{L}$ backprob $\hat{y}^{<t>}$ then calc $a^{<t>}$ & pass to $\hat{y}^{<t-1>}$, until $\hat{y}^{<1>}$.

6.3. Text Analysis using RNNs

Idea: use RNNs for seq gen (e.g. completing sentences).
- Need sequences and probs of terms to appear.
- Require large train set.



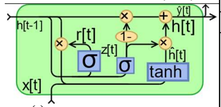
Train language model. Then, for each step, sample from SoftMax based on trained model prob.
Plug out of step t to in of step $t+1$.
When to stop?
1. Include <EOS> token in vocab, or
2. Use char RNN and stop when ".".

6.4. Long-Term Dependencies

Vanishing grad problem: RNN output influenced mostly by local inputs, so

it's hard to model: "the **boy**, who had …, **was** happy".
Solution: Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM)

6.4.1. Gated Recurrent Unit (GRU)

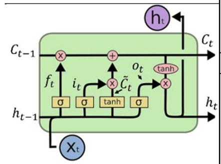


- $W^{(z)}$: input weights
- h_{t-1}: info passed from prev units
- $U^{(z)}$: update gate weights
- u_r: reset gate weights
- σ: sigmoid, ×: Hadamard prod

Update: decide what to remember.
$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$
- Update gate solves vanishing grad problem since it can decide to keep all history, thus eliminating vanishing grad

Reset gate: decide what to forget.
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$
Current Memory Content: decide what to keep by multiply reset gate and past info:
$$\hat h_t = \tanh(W\cdot x + r_t\odot U h_{t-1})$$
Final Memory at Current Time Step: info to keep is replaced by reset gate, and info to update is applied on past info.
$$h_t = z_t\odot h_{t-1} + (1-z_t)\odot \hat h_t$$

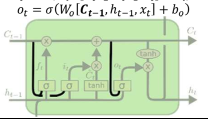
6.4.2. LSTM



Cell State (top line): info from past iters. Key element is that it can pass unchanged for multiple iters.
Forget Gate (low left σ):
$$f_t = \sigma(W_f\cdot[h_{t-1},x_t]+b_f)$$
– 1 ⇒ keep C_{t-1} "as is", 0 ⇒ discard
Input Gate (2nd low left σ and tanh): sigmoid decide which C_{t-1} values to update, tanh gen new values $\tilde C_t$
$$i_t = \sigma(W_i\cdot[h_{t-1},x_t]+b_i)$$
$$\tilde C_t = \tanh(W_C\cdot[h_{t-1},x_t]+b_C)$$
Update Old Cell State: forget parts then add only new info selected from $\tilde C_t$
$$C_t = f_t * C_{t-1} + i_t * \tilde C_t$$
Output Gate: sigmoid to decides cell state parts to output, then output chosen parts with tanh on cell state
$$o_t = \sigma(W_o[h_{t-1},x_t]+b_o)$$
$$h_t = o_t * \tanh(C_t)$$

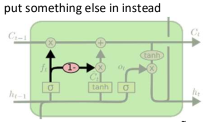
6.4.2.1.LSTM Variants: Peephole Connections

Idea: enables gates (sometimes not all) to take cell state into account
$$f_t = \sigma(W_f\cdot[\boldsymbol{C_{t-1}}, h_{t-1}, x_t]+b_f)$$
$$i_t = \sigma(W_i\cdot[\boldsymbol{C_{t-1}}, h_{t-1}, x_t]+b_i)$$
$$o_t = \sigma(W_o[\boldsymbol{C_{t-1}}, h_{t-1},x_t]+b_o)$$



6.4.2.2.LSTM Variants: Coupling Input and Forget Gates

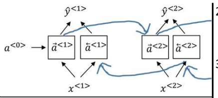
Idea: can only forget something if you put something else in instead



$$C_t = f_t * C_{t-1} + \boldsymbol{(1-f_t)} * \tilde C_t$$

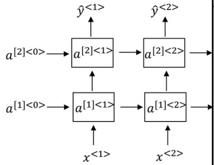
6.5. Bidirectional RNNs

Idea: analyze words appear after
Loss: $\hat y^{<t>} = g(W_y[\vec a^{<t>}, \overleftarrow a^{<t>}] + b_y)$
– Works well with GRU & LSTM.



6.6. Deep RNN

Idea: stack RNNs



$$a^{[2]<2>} = g\left(W_a^{[2]}[a^{[2]<1>}, a^{[1]<2>}] + b_a^{[2]}\right)$$
– Useful in time-series analysis for diff inputs arrive at diff intervals.

7. Word Embeddings (WEs)

Transfer Learn & WE: Learn/download large set WEs (1-100B) → transfer embeds to new task with smaller data set
– Common: embedding of size 300

7.1. Analogies

Idea: look for terms whose distance from target is like distance provided by example.
– Cosine similarity (common): angle between vecs
$$Sim(u,v) = \frac{u^Tv}{||u||_2||v||_2}$$
– Euclidean distance: also used.

7.2. Embeddings Matrix

Idea: mat that when mult by one-hot-dot rep of vocab returns embed.
– For vocab & embed size 10K & 300 ⇒ Embed mat $= 300\times10^4$ each column is embed for 1 term
Alg: to gen embed mat, random init mat then modify with grad des.

7.3. Creating WEs

7.3.1. NLP Models (complex)

Idea: predict next word (o_7) in seq o_j for $j\in\{1,\dots,6\}\to$ calc $e_j = E\cdot o_j$ for embed mat E & out SoftMax.
– NN params: W_1, b_1 (layer) and W_2, b_2 (SoftMax).
– Vars: use seq words before and after, use nearby words.

7.3.2. Word2Vec (skip-gram)

Idea: transfer to supervised problem.
Alg: for context word c, rand select target $t\in$ window (before/after) & mark distance of t from s.
Example: $|V| = 10^4$, context c, target t: $\hat y = \sigma(e_c) = \sigma(Eo_c) = SoftMax(Eo_c)$.
– Params optimized are only E, σ.
Loss: $L(\hat y, y) = -\sum_{i=1}^{10^4} y_i\log\hat y_i$ for $y,\hat y$ one-hot vecs of size 10^4.
SoftMax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$ for θ_t params of out term t.
Problem: takes too long. $\forall t$ requires performing 10^4 calcs
Possible solutions:
– Hierarchical model (eg B-tree, with freq terms closer to root)
– Negative sampling

7.3.3. Negative Sampling

Idea: select context word c then (1) rand select target t from sentence, mark True. (2) for same c, sample words $\in V$ mark False. Model as LogReg supervised problem:
$$P(y=1|c,t) = \sigma(\theta_t^T e_c)$$
– Note: word in sample can be rand selected and marked 0.
– More efficient than skip-gram: calc 1 pos $+k$ neg instead of 10^4

Options for selecting False samples:
1. Based on frequency in set ⇒ get many terms like a/an/the/…
2. Uniform sampling ⇒ doesn't represent set, may reduce representation effectiveness
3. Middle ground:
$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_i)^{3/4}}$ ⇒, this works best (empirically).

7.3.4. GloVe (Global Vec Word Rep)

Idea: accurately estimate prob of target t appear in context c.
Alg: $\min\sum_{i=1}^{10000}\sum_{j=1}^{10000} f(x_{ij})(\theta_i^T e_j + b_i + b_j - \log x_{ij})^2$
– x_{ct} = # of times t appeared in context c.
– rand sampling ⇒ $x_{ct} = x_{tc}$ (unlike seq word sample).
– Weighting term $f(x_{ij})$ prevents $\log 0$ where $f(0) = 0$ (we'll assume $0\log 0 = 0$)
– b_i, b_j are each term weights add
Problem: hard to rep "bigger pic" (eg 2 paragraphs, same words, diff mean)
Solution: Doc2Vec

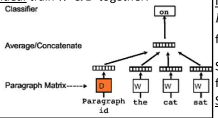
7.3.5. Doc2Vec: Doc Embedding

Idea: joint rep of term & paragraph.
Alg: use Word2Vec to map term to unique column $\in$ mat W
$p(w_t|w_{t-k},\dots,w_{t+k}; W) = \frac{e^{y_{wt}}}{\sum_i e^{y_i}} \Rightarrow y = b + Uh(w_{t-k},\dots,w_{t+k}; W)$, for b, U SoftMax params, and h is concat of relevant vecs from W. Also, map every paragraph to unique column $\in$ mat D. Thus, each term is rep by both W & D
– Only diff for y is that h is now concat of vecs from W & D

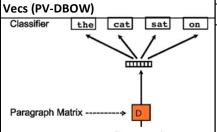
Approaches to train paragraph mat:
1. Distributed memory model of paragraph vecs (PV-DM)
2. Distributed Bag-of-Words of paragraph vecs (PV-DBOW)
3. Combine (i.e. concat) (1)+(2)

7.3.6. Distributed Mem Model of Paragraph Vecs (PV-DM)

Idea: train W & D together.



7.3.7. Distributed BoW of Paragraph Vecs (PV-DBOW)



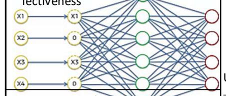
– More efficient since only store SoftMax weights

8. Autoencoders (AEs)

Idea: recon input by learning: (1) encoder (input → hidden layer), (2) decoder (hidden layer → out).
– Forces hidden layer to rep essence of input (like PCA)
– Makes model robust to noise
– Common: $\#n^{[hidden]} < \#n^{[input]}$ else some nodes are disabled for each train iter (sparse AE)

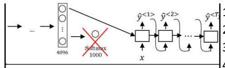
8.1. De-Noising Images (Masking)

Idea: recon corrupted images.
Alg: disable rand % of input ⇒ create holes in data ⇒ net forced to cope with it ⇒ more robust.
– FF nets work better despite CNN effectiveness



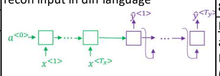
8.2. Image Captioning

Idea: use image classification CNN and replace final SoftMax layer with RNN that output caption.



8.3. Machine Translate (seq2seq)

Idea: build conditional lang model to recon input in diff language



– Note: better for reversed inputs.
Experimental Setting:
– 4-layer (10^3 nodes each) DLSTM
– 10^3 dim WEs.
– $|V_{input}| = 160\cdot10^5$; $|V_{out}| = 8\cdot10^4$
– 380M params (64M of RNN)
– $LR_{init} = 0.7$, after 5 epochs $LR_i = 0.5\cdot LR_{i-1}$
– Trained for 7.5 epochs on English to French set of 12M sentences
Alg: Find most likely translation $P(y^{<1>},\dots,y^{<T_y>}|x)$ for $y\in$English, $x\in$French:
$$\arg\max_{y^{<1>},\dots,y^{<T_y>}} P(y^{<1>},\dots,y^{<T_y>}|x)$$

8.3.1. Beam Search Algorithm

Idea: hold B highest prob options.
Alg: for each step, choose set with highest probs (in eg, for 2nd step $3\cdot10^3$ options), and finally, select top B options (i.e. 3)
– In each step all prev steps are considered, $P(y^{<1>},y^{<2>}|x) = P(y^{<1>}|x)P(y^{<2>}|x,y^{<1>})$
– Greedy Search (don't work well): when $B = 1$. Eg: "Jane is visiting…" vs "Jane is going…", 'going' is more likely than 'visiting', but overall less optimal.
Problems:
1. Numeric underflow
2. Prefer shorter translations

8.3.1.1.Numeric Underflow

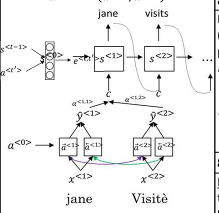
Idea: for most likely translation $P_t = P(y^{<t>}|x, y^{<1>},\dots,y^{<t-1>})$, ranking func is $\arg\max_y\prod_{t=1}^{T_y} P_t$.
Since $\forall t P_t \ll 1$, then we have underflow err.
Solution: use this rank func
$\arg\max_y\sum_{t=1}^{T_y}\log P_t$.

8.3.1.2.Prefer Shorter Translations

Idea: multiply of long prob seq lead to smaller values ⇒ unfair advantage to shorter translations
Solution: norm by # of words
$\frac{1}{T_y}\arg\max_y\sum_{t=1}^{T_y}\log P_t$
– Based on empiric testing

8.3.2. Attention Models

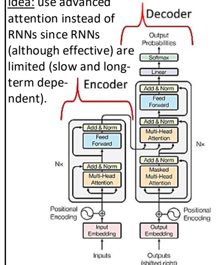
Idea: translate long seqs.
Decoder gets context: $c^{<i>} = \sum_{t'}\alpha^{<i,t'>}a^{<t'>}$, for $\alpha^{<t,t'>}$ is amount of attention $y^{<t>}$ should Pay to $a^{<t'>}$:
$$\alpha^{<t,t'>} = \frac{\exp e^{<t,t'>}}{\sum_{t'=1}^{T_x}\exp e^{<t,t'>}}$$
For t' is original French input
$$a^{<t'>} \equiv (\vec a^{<t'>}, \overleftarrow a^{<t'>})$$



Use SoftMax for $\sum_{t'}\alpha^{<t,t'>} = 1$
– Major shortcoming: takes quadratic time (but seqs are short, so not that bad)
Usage examples:
– Attention Models in Images (Also used for image captioning):
1. Input layer→
2. Conv Feature Extraction→
3. RNN with attention over image→
4. Word-by-word gen
– Date Translation: July 20th 1969→1969-07-20

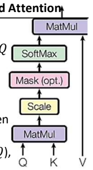
8.3.3. Transformer (tensor2tensor)

Idea: use advanced attention instead of RNNs since RNNs (although effective) are limited (slow and long-term dependent).



8.3.3.1.Scaled Dot-Prod Attention

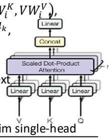
$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$, for Q queries (dec hidden state), K keys (enc hidden state), V values (relevance of enc hidden states [attention weights], based on K, Q), $d_k = \dim K = \dim Q$.



– Use scaling since dot-product values tend to grow with dim
– Enc uses source sentence's embed as Q, K, V.
– Dec uses enc's out for K, V, and its own out embed as Q.
– Note: enc & dec have multiple layers, so as above is correct for 1st layer. Other layers input is out of predecessors

8.3.3.2.Multi-Head Attention

Idea: for each attention head apply diff lin transform, then concat output and rescale. $MultiHead(Q,K,V) = concat(h_1,\dots,h_h)W^O$ for $h_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$, $W_i^Q, W_i^K\in\mathbb{R}^{d_{model}\times d_k}$, $W_i^V\in\mathbb{R}^{d_{model}\times d_v}$



– Help model multi dependencies in text (which is hard for single attention).
– Comp cost ≈ full dim single-head

8.3.3.3.Encoder (Enc)

$N = 6$ layers, each with 2 sub-layers:
1. Multi-head attention
2. FFN (not layer!) with residual connection and layer norm (like batch norm) between each layer

8.3.3.4.Decoder (Dec)

Like enc, but with masked attention heads, to disable future input

8.3.3.5.Tying Embeds & SoftMax

Same input & out embed weights improves performance & train time

8.3.3.6.Positional Encoding

No out-of-box input order modeling (since no use of CNNs/RNNs), so gen pos embed of each token:
$$PE(pos,i) = \begin{cases}\sin\left(\frac{pos}{10^{4\cdot\frac{2i}{d_{model}}}}\right), & i = 2k\\ \cos\left(\frac{pos}{10^{4\cdot\frac{2i}{d_{model}}}}\right), & i = 2k+1\end{cases}$$

8.3.4. OpenAI Gen Pre-Training (GPT)

Problem: not enough labeled sets of text (unlike unlabeled text)
Solution: transfer learning. Apply 2-step learning process from large unlabeled set, then fine-tune model for supervised learning problem
– Sets new state-of-the-art standards in 9/12 evaluated tasks.
Alg:
1. Unsupervised pre-train multi-layer Transformer-decoder to maximize $L_1(\mathcal{U}) = \sum_i\log P(u_i|u_{i-k},\dots,u_{i-1};\theta)$ for context window k, and θ net params (use word & pos embeds);
2. Supervised fine-tuning: add FF layer on top of Transformer-decoder, feed labeled sample $\{x_1,\dots,x_n\}$ to modified net, then new FF layer pred y.

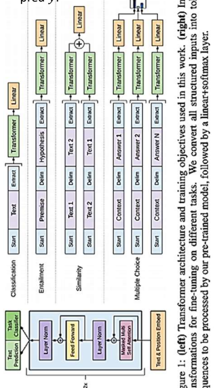


Figure 1: (left) Transformer architecture and training objectives used in this work. (right) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

FF layer loss function: $L_2(\mathcal{C}) = \sum_{(x,y)}\log P(y|x^1,\dots,x^m)$ for $x^1,\dots,x^m$ Transformer-decoder out (not original input $\{x_1,\dots,x_n\}$).
Overall loss: mix of L_1, L_2:
$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C})$$

8.3.5. ELMo (Embedding from Learning Models)

Idea: gen contextualized word embeds & do transfer learn. Considers not just token, but its context.
– Useful in cases like: "He kicked the **bucket**", "I have yet to cross-off all the items on my **bucket** list", "The **bucket** was filled with water".
Alg: use bi-directional lang models (biLMs) to pred next/previous word in sentence. Concat out to 2 FF layers. Out final prob with SoftMax.
– Prior biLM: use char-based embed (ID partial matches & supports stemming-like operation). Apply n-gram char conv with 2048 filters. Feed out to 2 highway layers.
– Effective at RNN input & out.

8.3.6. BERT Architecture

Idea: rather than train specifically per task, train on 2 general problems: (1) Masked language model (LM); (2) Next Sentence prediction.

8.3.6.1.Masked LM

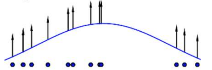
Idea: ∀ tokens seq, rand mask 15% and try to pred tokens (since its biLM ⇒ no use pred final terms). When token is chosen for replacement: 80% replace with [MASK], 10% replace with rand word, 10% do nothing.
– Rand word replacing makes it robust (can't try to pred it).
– Do nothing adds bias towards observed word.

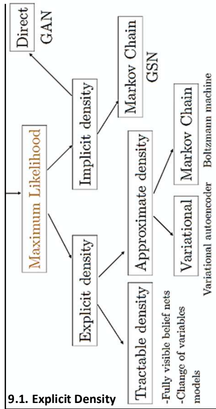
8.3.6.2.Next Sentence Pred

Idea: train binary clf to pred if 2 given sentences are subsequent.
– improved model's performance.

9. Lecture 5 – Generative (G) Model

Discriminative (D) model learn (hard/soft) **boundary** between classes, and **Generative (G)** learn individual classes **distribution**.
Maximum Likelihood: $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol\theta}\mathbb{E}_{x\sim p_{data}}\log p_{model}(\boldsymbol{x}|\boldsymbol{\theta})$





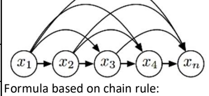
9.1. Explicit Density

Idea: If dist of data is known then def it explicitly (e.g. Gaussian).
– RNNs use explicit density

9.1.1. Tractable Density

Dist is tractable ⟺ ∀ marginal prob induced by it is comp in lin time. Tractable since they force some (arbitrary) order of values ∀ sample (eg image pixels are gen from top left to bottom right).

9.1.1.1.Fully Visible Belief Nets (Auto-Regressive Models)



Formula based on chain rule:
$$p_{model}(x) = \prod_{i=1}^{n} p_{model}(x_i|x_1\dots x_{i-1})$$
– Not necessarily apply only to seq data ⇒ can model vec as seq (like tabular sets often analyzed).
Cons:
1. $O(n)$ gen time (can't parallel since values must be gen in seq);
2. Hard to combine latent info (eg whole image state). Must use algs like varAE & PixelCNN

9.1.1.2.Change of Variables

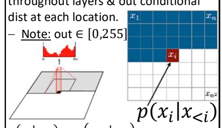
Idea: calc prob dist over y (latent variables dist), instead of over x.
$y = g(x)\to p_x(x) = p_y(g(x))\left|\det\left(\frac{\partial g(x)}{\partial x}\right)\right|$
– Eg non-lin Independent Component Analysis (ICA)
Cons: (1) g must be invertible; (2) latent & visible dim must match.

9.1.1.3.Autoregressive G Models

Commonly applied to seq data (eg language modeling, time series), not easy for non-seq data.
Alg: choose ordering of dims $\in X$ and def conditionals in prod rule:
$$p(x) = \prod_{k=1}^{N} p(x_n|x<k)$$
Pros: $p(x)$ is tractable⇒easy train & sample.
Cons: no natural latent rep

9.1.1.4.PixelCNN

Idea: use masked conv to pred next pixel. Preserves spatial res of input throughout layers & out conditional dist at each location.
– Note: out $\in[0,255]$

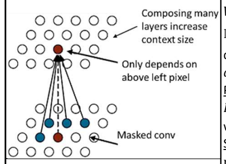


$p(x_i|x_{<i}) = p(x_{i,R}|x_{<i})\cdot p(x_{i,G}|x_{i,R},x_{<i})\cdot p(x_{i,B}|x_{i,R},x_{i,G},x_{<i})$

Teacher Forcing: Replace **blue** mat above with **red-green** mat. ie feed g-Truth regardless of prev preds

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

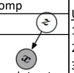
$\hat y$ ⇒ can train segments in parallel.
Problem: only works for training ⇒ slow test ⇒ doesn't scale well



9.1.2. Approximate Density

Comp expensive ⇒ must use approximations, else might can't finish comp

9.1.2.1.Variational AE



Var Learn: $p_{model}(x) = \int p_{model}(x,z)dz$
Problem: Latent models have intractable density ⇒ hard to calc integral directly ⇒ must rely on sampling (Monte Carlo).
Solution: use lower bound
Variational Bound: $\log p(x)\ge\log p(x) - D_{KL}(q(z)\|p(z|x)) = \mathbb{E}_{z\sim q}\log p(x,z) + H(q)$ for q is new dist over z, $\log p(x)$ is recon loss.
– Use KL divergence to force mean close to 0 and std var of 1. Force fit to predefined dist.
– $D_{KL}(P_r,P_g) = \int\log\left(\frac{P_r(x)}{P_g(x)}\right)\cdot P_r(x)d\mu(x)$, but can't calc it on 1 sample, so use dist rep instead.
– Want $q(z)$ to be close to z (perfect match ⇔ KL divergence = 0)
– Good value for bound⇒ even better value for $p(x)$
Idea: Instead finding optimal q params (like in past), let NN pred q.
Alg: def model for pred optimal q, def $p(z|x)$ via another net. Using 1 cost func & grad des, optimize nets.
– Prevents alternating between optimizing q & model

9.2. Implicit Density

Idea: Can't write $p_{model}(x)$, only have model that gen samples from dist. It's only interaction with dist.

9.2.1. Markov Chains – G Stochastic Nets (GSN)

Idea: Samples are gen iteratively, then stochastically updated & refined. After several iters, get diff independent samples.
– Markov chains are effective way to get samples from dist.

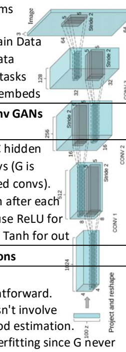
9.3. G Adversarial Nets (GANs)

Idea: train G & D against each other iteratively until G is better than D.
– Extends ad-reg train (see 3.10).
– Discriminator (D): pred if sample is real/fake.
– Generator (G): gen fake samples from rand noise that looks real (tries to fool D).
Alg: minibatch stochastic grad des train of GANs. k (hyperparm) is # of steps to apply to D,
\# Until G is better than D
For # of train iters:
  For k steps:
    \# Train D
    Gen m fake data by rand select noise samples $\{z^{(1)},\dots,z^{(m)}\}\in p_g(z)$ and insert to $G(z^{(i)})$
    Sample m Real examples $\{x^{(1)},\dots,x^{(m)}\}\in p_{data}(x)$
    \# Backprop D without changing G
    Update D by ascending its grad:
    $\nabla\theta_d\frac{1}{m}\sum_{i=1}^m\left[\log D(x^{(i)}) + \log\left(1-D\left(G(z^{(i)})\right)\right)\right]$
  \# Train G
  Sample m noise samples $\{z^{(1)},\dots,z^{(m)}\}\in p_g(z)$
  \# Backprop G without changing D
  Update G by grad desc:
  $\nabla\theta_g\frac{1}{m}\sum_{i=1}^m\log\left(1-D\left(G(z^{(i)})\right)\right)$
– Act as complicated loss.

GAN Objective (minimax game): D tries to max its reward $\boldsymbol{V(D,G)}$, G tries to min its reward (max loss)
$V(D,G) = \mathbb{E}_{x\sim p(x)}[\log D(x)] + \mathbb{E}_{z\sim q(z)}\left[\log\left(1-D(G(z))\right)\right]$ for data dist $P(x)$, implicitly def model dist $q(z)$.
Problem: If D is **too good** ⇒ $D(G(z)) = 0$ ⇒ grads = 0 ⇒ don't work well.
Solution: use diff formulation for G: $\min -\mathbb{E}_{z\sim q(z)}[\log D(G(z))]$
– More stable, but strong D can still cause problems
Usage:
1. Simulated Train Data
2. Fill missing data
3. Realistic gen tasks
4. Learn useful embeds

9.3.1. Deep Conv GANs (DCGANs)

Idea: replace FC hidden layers with convs (G is fractional-Strided convs). Add batch norm after each layer. Inside G use ReLU for hidden layers & Tanh for out



9.3.2. Pros & Cons

Pros:
– Gen is straightforward.
– Training doesn't involve Max Likelihood estimation.
– Robust to overfitting since G never sees train data.
– [Empirically] good at capturing modes of dist.

Cons:
– Prob dist is Implicit
  ▪ Comp $p(x)$ is not straight ⇒ Vanilla GANs (DCGAN) are only good for gen.
– Notoriously hard to train
  ▪ Non-convergence: D easily wins at start of training
  ▪ Mode-collapse: failure mode that learns just 1 image
  ▪ Hard to gen image diversity.
– Hyper-param selection is critical.

9.3.3. Non-Convergence

In general, DL models involve 1 player which tries to max its reward (ie min loss), uses SGD (with backprop) to find optimal params.
⇒ SGD guarantees convergence (under certain conditions).
Problem: with non-convexity, might converge to local optima $\min_G L(G)$.
GANs involve 2 (or more) players. D tries to max its reward; G tries to min D's reward $\min_G\max_D V(D,G)$
⇒SGD was not designed to find Nash equilibrium of game.
Problem: might not converge to Nash equilibrium at all. Eg:
$\min_x\max_y xy;\ \frac{\partial}{\partial x} = -y;\ \frac{\partial}{\partial y} = x$
$\frac{\partial^2}{\partial y^2} = \frac{\partial}{\partial x} = -y$
⇒ Solution has sinusoidal terms
⇒ Even with small LR, no converge

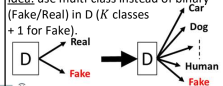
9.3.4. Mode-Collapse

9.3.4.1.Solution: Mini-Batch GANs

Problem: at Mode Collapse, G produces good samples, but very few⇒D can't tag them as fake.
Solution: let D know this edge-case by looking at batch instead of single examples. If lack diversity⇒mark examples as fake. Thus, G is forced to produce diverse samples.
Alg:
1. Extract features that capture diversity in mini-batch (eg L2-norm of diff between all pairs ∈ batch)
2. Feed features to D with image
3. Feature values will differ between diverse & non-diverse batches
4. ⇒D relies on those features for clf
5. This in turn: (a) forces G to match those feature values with real data, and (b) gen diverse batches.

9.3.4.2.Solution: Supervision with Labels

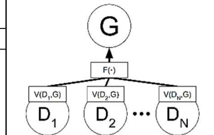
Idea: use multi-class instead of binary (Fake/Real) in D (K classes + 1 for Fake).



– Modify cross-entropy loss, s.t. it contains $K+1$ terms.
– No understanding why this works.

9.3.5. G Multi-Adversarial Nets

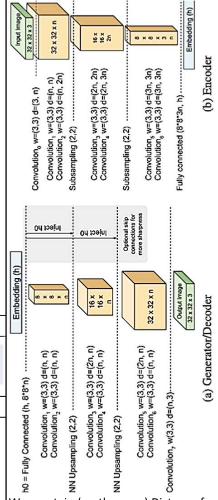
GAN problem: D init far better than G ⇒ D returns 0 for every gen sample ⇒ G can't improve
⇒ leads to "whack-a-mole" scenario: SoftMax makes everything sum to 1 ⇒ some params are reduced & others (relatively) increase. G tries something completely diff every time.



Formidable adversary: for F is max or boosting, results are bad (boosting not reported).
Forgiving teacher: make it easier for G to receive meaningful feedback (single D provides enough feedback).
Loss: $\frac{1}{N}\sum_i^N\mathbb{E}_{x\sim p_G(x)}\log(1-D_i(x))$
– Results show that more Ds (forgiving teacher) increases stability & convergence and reduces variance.

9.3.6. Boundary Equilibrium GAN (BEGAN)

Idea: use AE instead of classifier as decoder and calc diff between in recon dist (instead Fake/Real clf). Then, goal of D is to max this value, where one distribution is sampled from real data & other from generated.



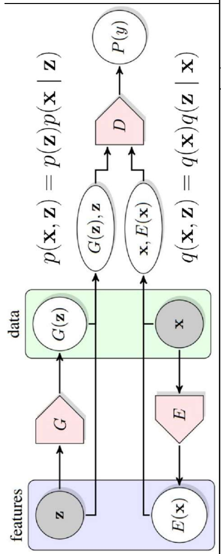
Wasserstein (earthmover) Distance for AEs: compares 2 loss distributions (not actual losses). It's the mean abs value

of diff between coupled samples drawn from both distribution:
$W_1(\mu_1,\mu_2) = \inf_{\gamma\in\Gamma(\mu_1,\mu_2)} \mathbb{E}_{(x_1,x_2)\sim\gamma}[|x_1 - x_2|]$
For $\mu_{1,2}$ two AE loss distributions, $\Gamma(\mu_1,\mu_2)$ all μ_1,μ_2 couplings set, and $m_{1,2} \in \mathbb{R}$ their respective means.
From Jensen's inequality:
Drive lower bound to $W_1(\mu_1,\mu_2)$:
$\inf \mathbb{E}[|x_1 - x_2|] \geq \inf|\mathbb{E}[x_1 - x_2]| = |m_1 - m_2|$
Loss: $\begin{cases}\mathcal{L}_D \\ \mathcal{L}_G\end{cases} = \begin{cases}\mathcal{L}(x;\theta_D) - \mathcal{L}(G(z_D;\theta_G);\theta_D), \text{for } \theta_D \\ -\mathcal{L}_D, \text{for } \theta_G\end{cases}$
Perfect equilibrium: $\mathbb{E}[\mathcal{L}(x)] = \mathbb{E}[\mathcal{L}(G(z))]$ for $\mathbb{E}$ is error.
Diversity bound: $\gamma = \frac{\mathbb{E}[\mathcal{L}(G(z))]}{\mathbb{E}[\mathcal{L}(x)]}$
D has two competing goals: (1) AE Real images, (2) separate Real images from Fakes. γ enables to balance goals:
1. Small $\gamma \Rightarrow \mathbb{E}(\mathcal{L}(x)) \gg \mathbb{E}(\mathcal{L}(G(z)))$, ie D focus on auto-encoding which makes G focus on image quality rather than diversity.
2. High $\gamma \Rightarrow \mathbb{E}(\mathcal{L}(x)) = \mathbb{E}(\mathcal{L}(G(z)))$, ie, D focus on separating which makes G focus on diversify to escape detection

Objective: $\begin{cases}\mathcal{L}_D \\ \mathcal{L}_G \\ k_{t+1}\end{cases} = \begin{cases}\mathcal{L}(x) - k_t\mathcal{L}(G(z_D)), \text{for } \theta_D \\ \mathcal{L}(G(z_G)), \text{for } \theta_G \\ k_t + \lambda_k\left(\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G))\right), \forall t\end{cases}$
t is train step, $\lambda = 10^{-3}$ us LR, $k_0 = 0$.
Convergence measure: $\mathcal{M}_{global} = \mathcal{L}(x) + |\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G))|$ based on Real data loss & diff between Real & Fake loss
- When ratio stabilizes⇒ can tell when to stop and when there's mode collapse.

Pros:
- Simple & robust, improves convergence.
- Equilibrium concept, balance G & D
- New convergence measure (when to stop train)
- Note: **can train D & G simultaneously** (and not in turns).

9.3.7. Bidirectional GAN (BiGAN): Adversarial Feature Learning

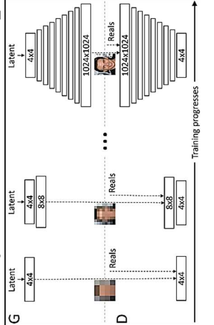


Problem: GANs don't provide a way to gen meaningful latent space representations (only gen samples), but such reps can be more useful than samples.

Solution: add AE E (in addition to G & D), s.t. D receives $[(x, E(x)), (G(z), z)]$ (instead of $(x, G(z))$). Thus, to min loss E needs to learn to invert G.
- Note: the need to gen high-quality embeds also improves overall performance.

Train objective: a minimax game $\min_{G,E}\max_D V(D,E,G)$ for $V(D,E,G) \equiv$
$\mathbb{E}_{x\sim p_x}\left[\underbrace{\mathbb{E}_{z\sim p_E(\cdot|x)}[\log D(x,z)]}_{\log D(x,E(x))}\right] + \mathbb{E}_{z\sim p_z}\left[\underbrace{\mathbb{E}_{x\sim p_G(\cdot|z)}[\log(1 - D(x,z))]}_{\log(1-D(G(z),z))}\right]$

9.3.8. Progressive Growing of GANs

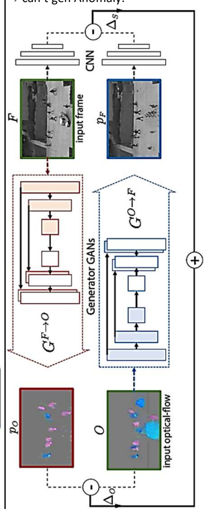


Idea: hard to gen high-quality images with GANs. Easier to train for low res & fine-tune. Grow G & symmetrically. Residual connections that gradually phased out makes train smooth. Also, no need to "freeze" layers.
- Makes train faster (×2-6) & more stable.

Scaling and normalization:
(Common) Problem: competition between G & D causes grads to spiral out of control
Solution: batch-norm
Problem: batch-norm scales based on standard deviation, making update independent of scale (ie updates can be both too big **and** too small).
Solution: instead, scale all weights at runtime, using 1 norm const per layer. ⇒ All activations are of the same scale and no params update is required (unlike batch-norm).

9.3.9. Anomaly Detection GAN

Idea: train GAN on Normal data only ⇒ can't gen Anomaly.



10. Lec 7: Recommender Systems DL (RecSys)

Why recommender system is an interesting ML problem?
- Bipartite graph – pred match
- Ranking problem
- Sparse data ("dirty" labels)
- Missing relevant info
- Pred human behavior
- Scale

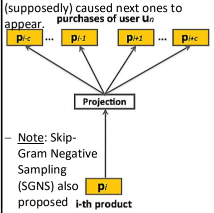
Pros of DL for RecSys:
- Extract features directly from content
- Easy handling of heterogenous data
- Dynamic/sequential behavior modeling with RNNs
- More accurate representation learn of users & items (natural extension of CF & more)

10.1. Prod2Vec: Embedding for Recsys

Idea: replace words in Word2vec (see 7.3.2) with items in user session/ profile (Item = Word, User's Item Clicks = Document) (eg, user shopping sessions). Resulting embed co-locate products that are close to each other.

10.1.1. Skip-Gram Prod2Vec

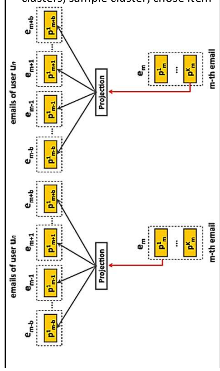
Idea: input i[th] product user purchased, context is other purchases of user.
Alg: use text of email receipts to ID items & seq ⇒ products that appear in similar seqs have similar contexts.
Problem: Multiple items in same receipt are confusing, because it's harder to know which item (supposedly) caused next ones to appear.



- Note: Skip-Gram Negative Sampling (SGNS) also proposed

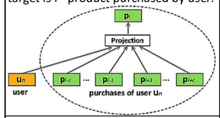
10.1.2. Bagged Prod2Vec

Idea: operate at email level rather than product level. Input products user purchased in 1 basket (basket is sum of products embeds), context is other user baskets.
Clusters select follow multinomial dist, s.t. more probable clusters have higher chance to be chosen. Once cluster is chosen, sort items based on embed cosine similarity & chose top K.
Loss: $\mathcal{L} = \sum_{s\in S}\sum_{e_m\in s}\sum_{-n\leq j\leq n; j\neq 0}\sum_{k=1,\dots,T_m}\log\mathbb{P}(e_{m+j}|p_{mk})$, ie sum of probs of observing products from neighbor emails over emails seq, for product k & email m.
2 recommendation approaches:
1. topK: select k most similar items
2. Clustering: use K means to create clusters, sample cluster, chose item



10.1.3. User-Prod2Vec

Idea: learn user rep (like paragraph2vec). User embeds added as global context. Input user and products purchased except for the i[th]; target is i[th] product purchased by user.



10.1.4. Meta-Prod2Vec

Problem: Prod2Vec considers adjacent items doesn't, but not items metadata (ie categories, origin country, …) and doesn't offer way to combine it.
Idea: feed item & context metadata with embed & pred **both**.
- Loss function cross entropy

10.1.5. Content2Vec

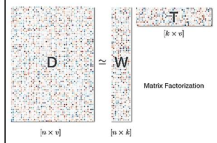
Idea: Separate modules for multi-model info [eg Prod2Vec (CF), AlexNet (Image), Word2Vec & TextCNN (Text)] and learns pairwise similarities (likelihood of buying two items together.

10.2. Deep Collaborative Filtering

Idea: pred top N recommendation (eg, recommend movies)

10.2.1. Dimensionality Reduction

Problem: too many features.
Solution: Matrix Factorization (low rank approximation to original matrix)
$\min_{v_u,q_i,b_u}\sum_{r\in R}(r_{uis} - \vec{v_u}\cdot\vec{q_i} - \bar{\imath} - b_u)^2$
for True Rating, Pred.



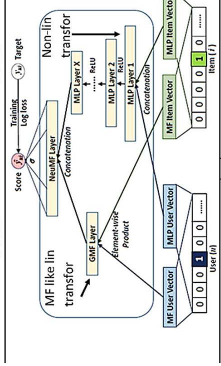
Other Solutions (dim reduction):
1. Singular Value Decomposition (SVD)
2. **Lin formula for pred**

10.2.2. AE for RecSys

Idea: recon corrupt user interaction.
Variants:
1. Collaborative DL (CDL) uses Bayesian stacked denoising AEs and tags\metadata instead of item ID
2. Collaborative Denoising AE (CDAE) uses additional user node on input and bias node beside hidden layer

10.2.3. Neural Collaborative Filtering

3 collaborative filtering models:
1. Generalized Matrix Factorization (GMF)
2. Multi-Layer Perceptron (MLP)
3. Neural Matrix Factorization (NeuMF)
- Optimize using Log loss with negative sampling (for implicit feedback & ranking task).



Alg: given 2 separate sparse vecs (user & item), emed each twice & concat. Feed 1 to DNN & 1 to GMF.
- Using 2 embed sets allow more flexibility since **each approach can learn its own embed**.

Other concerns (hyper-params): embed size, sampling, # of layers, layers size, reg (dropouts, L2), batch size, LR, early stop, loss function, initializers, activations, "pre training".

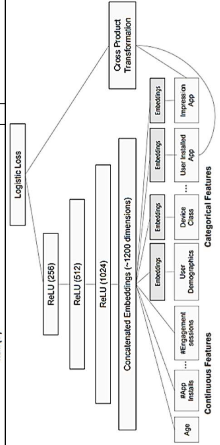
10.3. Memorization vs Generalization

RecSys try to achieve both:
1. Memorization: learn frequent items or features co-occurrence & exploit correlation available in history data.
   - Pros:
     - **Topical & directly relevant** recs
     - Effectively implemented with lin models (eg SVM)
     - Hand-crafted features work well
   - Cons:
     - Limited transferability
2. Generalization: based on correlation transitivity & explores new feature combinations that never (or rarely) occurred in past.
   - Pros:
     - **Improve recs diversity**
     - Better achieved using embeds
     - Can model latent connections among items
   - Cons:
     - Embeds are dense ⇒ many non-related items have similarity > 0 ⇒ lead to wrong recs (ie over-generalization).

10.4. Wide & Deep Learn for RecSys

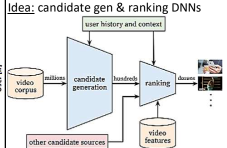
Idea:
- Wide: LogReg ($y = w^T x + b$) using raw & transformed features such as **cross-product transform** (1 only if all participating features are not 0) which adds additional non-lin & assists performance.
- Deep: FFN with embed rep of raw features, in which dense layer apply $a^{(l+1)} = f(w^{(l)}a^{(l)} + b^{(l)})$ with ReLU activation. Dense raw features are concat to embeds of sparse ones

Deep works better than wide (+2.9%), but deep & wide works best (+3.9%)



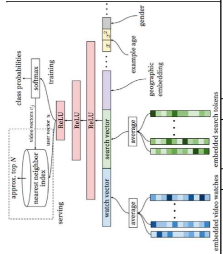
10.5. YouTube RecSys DNN

Idea: candidate gen & ranking DNNs



Problems:
1. Scale (very big)
2. Freshness (balance new vs well-established data)
3. Noise (no explicit feedback, diff to know if user liked video)
4. Rec must be made in **milliseconds**

10.5.1. Candidate Gen Component

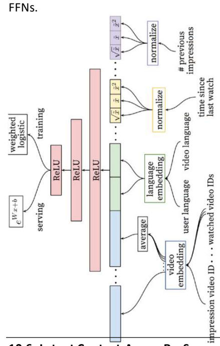


Idea: treat problem as **extreme multi-class** (ie rec specific movie at specific time), use fixed size video embed vec & sparse view user embed vec (transformed to dense embed form), jointly learn embeds with rest of net. Train with Negative Sampling (hierarchical SoftMax didn't produce good results).
To improve performance:
1. Limit views # allowed per user to make sure that small highly active users group doesn't bias data.
2. Consider all video views, including views from other websites, s.t. can learn about user interests not found via their RecSys
- Need to withhold some info from mode since if user search something, no point in making search results appear on home page immediately afterwards.
- Both width (# of features) & depth (# of layers) improved performance
- Video Age Effect: has significant effect on score given for 1[st] 10 days, but effect is gone after 20 days.

10.5.2. Ranking Component

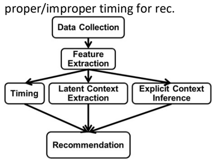
Time Watch eval metric: measure using weighted LogReg, specifically developed for this purpose (only used in train). Helps negate "click-bait" videos effect.
- Access much more info (possible since use smaller # of videos).

Problem: have many features. Especially for temporal features using FFNs.



10.6. Latent Context-Aware RecSys

Idea: input user context features to classifier to pred if current context is proper/improper timing for rec.



Latent Context-Rating Pred Rule (Extension): $\hat{r}_{u,i,c_1,\dots,c_k,l_1,\dots,l_d} = b_u + b_i + v_u q_i^T + \sum_{j=1}^{k} b_{ic_j} + \sum_{j=1}^{d} b_{ij} l_j$ for $b_{u,i}$ baseline estimator for user u and item i; v_u, q_i MF latent factor vecs for user u and item i; b_{ic_j} rating bias of item i under explicit context condition c_j; b_{ij} rating bias of item i for j^{th} latent context attribute; l_j latent context attributes.
Loss: MSE between pred (using pred rule) and actual rating.

10.7. AE for Feature Modeling

Idea: use hidden layer values as RecSys input. 1 rep for both original & latent context variables.

10.8. Session based Recommendation

Idea: anonymous user rec is common in real-life (occasional user, privacy). Use events seq (eg pred next click, intent).
Problem: classic algs can't cope well with item to item recs in live systems

10.8.1. Feature-Rich Session Rec

Idea: items have rich feature reps (eg pics & text descriptions). Use several parallel RNN (p-RNN) to model sessions based on clicks & features. GoogLeNet used for image enc.
Parallel nets train strategies:
1. Simultaneous: train all at once (baseline)
2. Alternating: train subnets in alternating fashion each epoch
3. Residual: train subsequent nets on errors of their predecessors. Train each net for long time (10 epochs)
4. Interleaving: use residual, but switch net every mini batch.

11. Robustness in DL

Areas of technical AI safety

11.1. Specification

Idea: define system purpose
1. Ideal: description that perfectly fits human designers wishes (what every specification should be).
2. Design: blueprint used to build sys (eg reward in reinforcement learn)
3. Revealed (behavior): what can be inferred (ie reverse engineer) from what really happens (eg look at agent behavior & infer reward func)

Areas:
1. Design: bugs & inconsistencies, ambiguities, side-effects, high-level specification languages, preference learn, design protocols
2. Emergent: wire heading, delusions, meta learn and sub-agents, detect emergent behavior

11.1.1. Faulty Reward Functions

Idea: Important to def reward func correct & make sure model learns from right sources:
- CycleGAN for converting aerial image to street maps & back steganographically enc output info in intermediary image without it being humanly detectable.
- Robotic arm trained to slide block to target position on table achieves goal by moving table instead.

11.2. Robustness

Idea: ensures AI sys continue to operate within safe limits upon problems
Problems: adversarial, dist shift, unsafe exploration.
Solutions: prevention (learn to avoid problematic scenarios), recovery (correct course after problem occurred)
Areas:
1. Prevention & risk: risk sensitivity, uncertainty estimates, safety exploration, cautious generalization, verification, adversaries
2. Recovery & stability: instability, error-correction, failsafe mechanisms, distributional shift, graceful degradation.

11.3. Assurance

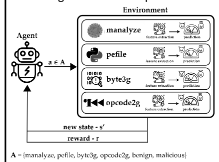
Idea: once AI sys is deployed, need to continuously monitor & adjust it.
Solutions:
1. Monitoring: inspecting sys to analyze & pred its behavior (including interpretability)
2. Enforcement: designing mechanisms to control & restrict sys behavior (including interpretability)

Areas:
1. Monitoring: interpretability, behavioral screening, activity traces, estimates of causal influence, machine theory of mind, tripwires & honeypots
2. Enforcement: interruptibility, boxing, authorization sys, encryption, human override.

11.4. Confidence in ML Classifications

Problem: SoftMax is poor confidence estimator & blunt instrument.
Example: many organizations setup pipeline to filter incoming files. Multiple commercial detectors filter all files. A heuristic (eg, score averaging) or ML alg is used to determine which files are blocked.
Problems with example:
1. Confidence score thresh doesn't model uncertainty.
2. Unclear how to treat errs (FP, FN) diff based on thresh
3. Small changes to thresh may cause problems in organizational policy

Solution: Bayesian DL analyze weights dist to offer uncertainty estimates, but its comp expensive.
Better solution: use reinforcement learns to guide detection process



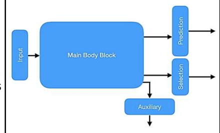
11.4.1. Dropout as Bayesian Approximation

Idea: dropout can be interpreted as approximation of well-known prob (Gaussian). Ie, info that so far has been thrown away can now be used to model uncertainty. Quantifying uncertainty & using it to tune model, improves performance of many tasks (clf, regression, reinforcement learn).
Alg: Run each sample multiple (eg hundreds) times, use KL divergence to measure (dis)similarity of output dist
Loss: clf error + model uncertainty
- Considered **Monte Carlo** process (sampling from dist) since dropout is done rand
- Apply both during train & test ⇒ slower than standard NN
- Note: net itself doesn't changes ⇒ can use for any DNN

11.4.2. SelectiveNet DNN with Integrated Reject Option

Idea: directly integrate an "I don't know" component into sys to enable it to reject classifying some samples Train component with net (simpler than prev solutions)
Selective model: is a pair (f,g) where f is **pred** func & $g: \mathcal{X} \to \{0,1\}$ is **selection** func.
$(f,g)(x) \equiv \begin{cases} f(x), & \text{if } g(x) = 1 \\ \text{unkown}, & \text{if } g(x) = 0\end{cases}$
Model performance (**coverage** & **risk**): $\phi(g) \equiv \mathbb{E}_P[g(x)]$, which is prob mass of un-rejected region
(Selective) Risk: $R(f,g) \equiv \frac{\mathbb{E}_P[\ell(f(x),y)g(x)]}{\phi(g)}$
- Clear trade-off between metrics

Optimal selective model: given desired coverage rate $0 < c < 1$: $\theta^* = \arg\min_{\theta\in\Theta}(R(f_\theta, g_\theta))$ s.t. $\phi(g_\theta) \geq c$
For θ DNN params.



Alg: main body (regular architecture) has 3 outputs:
- If selection > 0.5, classify. Else reject.
- Auxiliary output (h) only used in train & used to clf **all** samples
- Note: aux is important, since without it, net will only focus on small subset that it can easily classify.

Loss for f,g is: $\mathcal{L}(f,g) = \hat{r}_\ell(f,g|S_m) + \lambda\Psi\left(c - \hat{\phi}(g|S_m)\right)$
$\Psi(a) = \max(0,a)^2$ for λ param, Ψ quadratic penalty func.
Loss for aux: $\mathcal{L}(h) = \hat{r}(h|S_m) = \frac{1}{m}\sum_{i=1}^{m}\ell(h(x_i), y_i)$
Combined loss: $\mathcal{L} = \alpha\mathcal{L}(f,g) + (1-\alpha)\mathcal{L}(h)$

11.4.3. Deep K-Nearest Neighbors

Idea: solve 3 problems:
1. Confidence: not only produce clf, but also its certainty.
2. Interpretability: explain why pred was made
3. Robustness

Conformal pred: analyze classifier output to determine confidence in score. This is comp prohibitive (re-train model for each sample).
- Inductive conformal pred (more common): train on val set

Non-conformity measure: how diff is one sample from prev samples in same dist. For sample x with label j:
$$\alpha(x,j) = \sum_{\lambda\in 1,\dots,l}|i \in \Omega_\lambda : i \neq j|$$
- Natural candidate: neighbors in adjacent clusters
- Alg may assign same score for diff reasons (1 example can get certain score since others like it was seen before & alg is extrapolating; example is ambiguous, possibly due to adversarial attack)

Empirical p-value: find % of samples with higher scores $p_j(x) = \frac{|\{\alpha\in A : \alpha \geq \alpha(x,j)\}|}{|A|}$ for val set A, label j.
Alg: for train data (X,Y), calibration data (X^c, Y^c), train net f with l layers, k # of neighbors, test input z:
\# Comp layer-wise k-NN for z
For each layer $\lambda \in 1,\dots,l$:
  $\Gamma = k$ points in X closest to z found with LSH tables
  $\Omega_\lambda = \{Y_i : i \in \Gamma\}$ # k labels found
\# Comp pred, confidence, credibility
\# Calibration:
$A = \{\alpha(x,y) : (x,y) \in (X^c, Y^c)\}$
For each label $j \in 1,\dots,n$:
  \# Non-conformity:
  $\alpha(z,j) = \sum_{\lambda\in 1,\dots,l}|i \in \Omega_\lambda : i \neq j|$
  \# empirical p-value
  $p_j(x) = \frac{|\{\alpha\in A : \alpha \geq \alpha(z,j)\}|}{|A|}$
pred $= \arg\max_{j\in 1,\dots,n} p_j(z)$
confidence $= 1 - \max_{j\in 1,\dots,n;\ j\neq pre} p_j(z)$
credibility $= \max_{j\in 1,\dots,n} p_j(z)$

11.4.4. ReLUplex

Idea: method for **complete verification** of DNN (but for small nets). Uses simplex-based methods for analyzing all possible constraints & ID regions where wrong outcome can be achieved. Algorithm def ranges for specific hyperparams.
- Can provide actual values that "break" model, s.t. user can provide diff input to resolves inconsistency

Example: To satisfy $v_{11} \in [0,1]$, $v_{31} \in [0.5,1]$, def: $a_1 = -v_{11} + v_{21}^b$, $a_2 = v_{11} + v_{22}^b$, $a_3 = -v_{21}^f - v_{22}^f + v_{31}$

variable	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
lower bound	0	$-\infty$	0	$-\infty$	0	0.5	0	0	0
assignment	0	0	0	0	0	0	0	0	0
upper bound	1	∞	∞	∞	∞	1	0	0	0

Outcome: $v_{11} = v_{21}^b - a_1$, $v_{22}^b = v_{11} - a_2$, $v_{21}^f = -v_{22}^f + v_{31} - a_3$

variable	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
lower bound	0	$-\infty$	0	$-\infty$	0	0.5	0	0	0
assignment	0.5	0.5	0.5	-0.5	0	0.5	0	0	0
upper bound	1	∞	∞	∞	∞	1	0	0	0

Problem: ReLUplex is very inefficient⇒ Can only apply to very small nets
Solution: DeepSafe

11.4.5. DeepSafe

Idea: use clustering to ID regions in input space, then apply ReLUplex only to these regions. If counterexample is found⇒ use it to reshape region. Else region safe.
- Approach doesn't provide absolute protection, but can ID regions in space in which one can operate freely
- Label-guided clustering: since homogenous regions are needed, clustering process is performed hierarchically by labels (in almost any other study ignore labels).
- Clusters are re-split until only 1 type of label remains
- Uses ReLUplex to analyze clusters to try to solve constraint: $\exists x. \|x - \text{cen}\|_{L_1} \leq r \wedge \text{score}(x,l') \geq \text{score}(x,l)$
  This means that inside cluster there is sample whose weight is higher for diff label than intended.
  - If this is the case, region can be reshaped to create (smaller) sage region(s)
- Diff distance metrics can be used to create & reshape clusters (Euclidean, Manhattan)

12. Lec 9: Graph Embed

Challenges:
1. Property Choice: selecting graph properties to preserve: distance metrics (directed/undirected), set of features for each graph (node features, edge features)
2. Scalability: some graphs may easily reach hundreds millions nodes, especially if interested in preserving global properties
3. Embed dim: often trade-off between rep richness and run time. Rich embeds can harm (relatively simple) like link pred, if only local connections are needed.

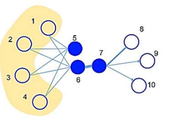
Graph: $G = (V,E)$, where V is vertices set that rep data object, E is set of edges between vertices, each rep relation.
- Edge $e \in E$ is ordered pair $e = (u,v)$, associated with weight $w_{uv} > 0$ that indicate connection strength
- If G is undirected $\Rightarrow (u,v) = (v,u)$ & $w_{uv} = w_{vu}$
- If G is directed $\Rightarrow (u,v) \neq (v,u)$ & $w_{uv} \neq w_{vu}$

12.1. 1[st] & 2[nd] order Proximity

1[st] order proximity: local pairwise proximity between two vertices.$\forall u,v \in V$ linked by edge $(u,v) \in E \Rightarrow$ edge weight w_{uv} indicates 1[st] order proximity.
- If $(u,v) \notin E \Rightarrow$ their 1[st] order proximity is 0

Fig: edges can be undirected, directed and/or weighted. v_6, v_7 should be placed closely in low-dim space as they are connected via string edge. v_5, v_6

should also be placed closely as they share similar neighbors.



2nd order proximity: proximity between two vertices (u,v) is similarity of their neighborhood net structures. Let $p_u=(w_{u,1},\dots,w_{u,|V|})$ be 1st order proximity of $u\in V$ with all other vertices ⇒ 2nd order proximity is determined by similarity of p_u & p_v.

12.1. Large-scale Info Net Embed (LINE)

Idea: preserve 1st & 2nd order proximity (not necessarily DL). Embeds for each proximity are calc separately & then concat.

Optimize using Negative Sampling or (edge based) Weighted Sampling.

12.1.1. 1st Order Proximity

Idea: prob of each undirected edge (i,j) to exist between vertices v_i, v_j is: $p_1(v_i,v_j)=\frac{1}{1+\exp(-\vec{u}_i^T\vec{u}_j)}$, for $\vec{u}_i\in\mathbb{R}^d$ is low dim rep of vertex v_i.

To preserve: min O_1 = min $d(\hat{p}_1(\cdot,\cdot),p_1(\cdot,\cdot))$ for d distance between two dists.

- KL-divergence is common option for d: $O_1=-\sum_{(i,j)\in E}w_{i,j}\log p_1(v_1,v_j)$
- Note: doesn't apply to directed graphs

12.1.2. 2nd Order Proximity

Idea: assumes vertices that share connections to other vertices are similar ⇒ Each $v_i\in V$ is rep by

1. $\vec{u}_i$ when treated as vertex (ie itself)
2. $\vec{u}_i'$ when treated as context for other vertices

Prob of context v_j gen by vertex v_i:

$$p_2(v_j|v_i)=\frac{\exp(\vec{u}_j'^T\cdot\vec{u}_i)}{\sum_{k=1}^{|V|}\exp(\vec{u}_k'^T\cdot\vec{u}_i)}$$

To preserve: contexts approximated dist must be like sampled dist: $O_2=\sum_{i\in V}\lambda_i d(\hat{p}_2(\cdot|v_i),p_2(\cdot|v_i))$ for d is distance between dists.

- If λ is analyzed vertex deg & choose KL-divergence as distance measure: $O_2=-\sum_{(i,j)\in E}w_{i,j}\log p_2(v_j|v_i)$

12.2. Node2Vec

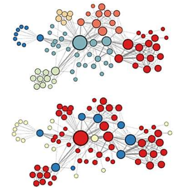
Idea: rand walks-based alg, but more robust than DeepWalk.

Uses broader 2nd order proximity def: not just sharing same neighbors, but having same roles (ie, neighborhood topology)

- Capable of dynamically alternating between preserving 1st & 2nd order proximity

Balance 2 similarities for embed nodes closely together:

1. Homophily: strongly interconnected nodes of same clusters.
2. Structural: nodes with same structure in net (eg hubs).



- Rep similarity with BFS & DFS.

Objective: max prob of observing neighborhood N_S for vertex u given its feature rep f

$$\max_f\sum_{u\in V}\log\Pr(N_S(u)|f(u))$$

To simplify rep use assumption:

1. Conditional independence: every observation of neighborhood vertex is independent of other vertices $\Pr(N_S(u)|f(u))=\prod_{n_i\in N_S(u)}\Pr(n_i|f(u))$
2. Feature space symmetry: source & neighborhood vertices have same effect on each other

Simplified objective: $\max_f\sum_{u\in V}\left[-\log Z_u+\sum_{n_i\in N_S(u)}f(n_i)\cdot f(u)\right]$ for $Z_u=\sum_{u\in V}\exp(f(u)\cdot f(v))$

- Note: use Neg Sampling since Z_u is very expensive to comp

Fixed length l rand walks: $P(c_i=x|c_{i-1}=v)=\begin{cases}\pi_{vx}/Z, & \text{if }(v,x)\in E\\ 0, & \text{else}\end{cases}$ for π_{vx} transition prob (weight), Z norm const.

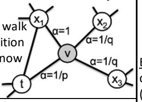
Bias Control walk (like BFS of DFS): $\alpha_{pq}(t,x)=\begin{cases}1/p, & \text{if } d_{tx}=0\\ 1, & \text{if } d_{tx}=1\\ 1/q, & \text{if } d_{tx}=2\end{cases}$ for d_{tx} shortest path between $t,x\in V$

Transition prob: $\pi_{vx}=\alpha_{pq}(t,x)\cdot w_{vx}$

Return param p: high value makes revisiting nodes more likely (like BFS).

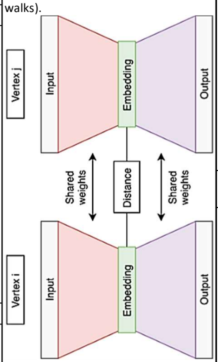
In-out param q: If $q>1$ ⇒ rand walk biased to visit nodes close to t. If $q<1$ ⇒ opposite.

Fig show rand walk that just transition from t to v & now eval next step from v.



12.3. Structural Deep Net Embed (SDNE)

Idea: use AE for 1st & 2nd order proximities (unlike node2vec rand walks).



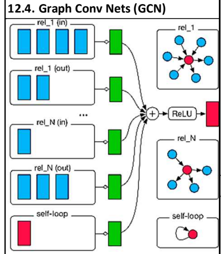
Loss for 2nd order proximity: recon err for each vertex neighborhood $\mathcal{L}_{2nd}=\sum_{i=1}^n\|(\hat{x}_i-x_i)\odot b_i\|_2^2=\|(\hat{X}-X)\odot B\|_F^2$ for $b_i>1$ for existing edges, and 1 for non-existing edges

Loss for 1st order proximity: for each vertex, distance from its connected vertices $\mathcal{L}_{1st}=\sum_{i,j=1}^n s_{i,j}\|y_i-y_j\|_2^2$

- Idea: connected nodes should have similar embed ⇒ penalized large diff in embed

Final recon loss: $\mathcal{L}=\mathcal{L}_{1st}+\alpha\mathcal{L}_{2nd}+\nu\mathcal{L}_{reg}$ for reg is regularization.

12.4. Graph Conv Nets (GCN)



Idea: stack convs for 1st & 2nd order proximities.

1st order proximity: convs of $v\in V$

2nd order proximity: conv of neighboring (connected) vertices

- Diff convs "in"/"out" connections.
- Use self-loops in all vertices s.t. one doesn't "lose itself" while going through stacked layers

Hidden state of layer $l+1$ for incoming nodes: $h_i^{(l+1)}=\sigma\left(\sum_{m\in M_i}g_m\left(h_i^{(l)},h_j^{(l)}\right)\right)$

Alg: concat all conv activations, then apply non-lin (ReLU) & pass result to next layer

12.4.1. Vertex Classification

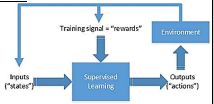
Idea: use convs as enc. Apply SoftMax to final layer to out possible labels $\mathcal{L}=-\sum_{i\in Y}\sum_{k=1}^F t_{ik}\ln h_{ik}^{(L)}$

12.4.2. Link prediction

Idea: use enc-dec to recon original net with additional links. $\mathcal{L}=-\frac{1}{(1+\omega)|\hat{E}|}\sum_{(s,r,o,y)\in\mathcal{T}}y\log l(f(s,r,o))+(1-y)\log\left(1-l(f(s,r,o))\right)$

13. Lec 10: Deep Reinforcement Learn

Idea:



Effective in sparse rewards [since don't need feedback for every move (but still need some reward to learn)]:

- trajectories form data
- To make seq of (related) decisions
- Observe (partial, noisy) feedback for actions choice
- Tasks require learning & planning.

RL challenges:

1. Access env: static set not enough since need to understand how actions affect env. For static set, need data that cover all possible situations (hard)
2. Jointly learning & planning from correlated samples
3. Data dist changes with action choice

13.1. Markov Decision Processes (MDP)

Idea: for action $a\in A$ in state $s\in S$ get reward $R(s,a)\in\mathbb{R}$

- State (= env) is discrete/continuous
- Action (= state transition) have prob $P(s'|s,a)\le 1$ (=1 if discrete)
- Finite MDP ⇒ final S,A,R

Markov Property: dist over future states depends only on present state & action $Pr(s_{t+1}|s_t,a_t)=Pr(s_{t+1}|s_1,a_1,s_2,a_2,\dots,s_t,a_t)$

- Simplify problem (eg traffic light) but not always true (eg blackjack)
- Critical for reward calc: $p(s',r|s,a)=Pr(s_t=s',r_{t+1}=r|s_t=s,a_t=a)$
- Hold: $\forall s\in S,a\in A\Rightarrow\sum_{r\in R}r\sum_{s'\in S}p(s',r|s,a)=1$

Expected reward of state-action: $r(s,a)=\mathbb{E}[R_{t+1}|S_t=s,A_t=a]=\sum_{r\in R}r\sum_{s'}p(s',r|s,a)$

Expected reward of transition to next state s' from current state s: $r(s',s,a)=\mathbb{E}[R_{t+1}|S_{t+1}=s',S_t=s,A_t=a]=\sum_{r\in R}r\frac{p(s',r|s,a)}{p(s'|s,a)}$

Objective: max expected return $G_t=\begin{cases}\sum_{t+1}^{T}R_k, & \text{Finite horizon}\\ \sum_{k=0}^{\infty}\gamma^k r_{t+k+1}, & \text{Infinite horizon}\end{cases}$ for discount factor $\gamma\in[0,1]$ (usually close to 1).

- Infinite idea: future rewards are less valuable than current rewards

13.2. Policies, Value Funcs, Q-Funcs

Policy (strategy) $\pi:S\to A$: action to do in state: $\pi(s,a)=P(a_t=a|s_t=s)$

- Goal: max value func

Value func: cumulative expected return of following π until final time T: $v_\pi(s)=\mathbb{E}_\pi[G_t|S_t=s]=\mathbb{E}_\pi[\sum_{k=0}^\infty\gamma^k R_{t+k+1}|S_t=s]$

- Assess how good is state s

Q-func: taking action a at state s, then follow policy π: $q_\pi(s,a)=\mathbb{E}_\pi[G_t|S_t=s,A_t=a]=\mathbb{E}_\pi[\sum_{k=0}^\infty\gamma^k R_{t+k+1}|S_t=s,A_t=a]$

- Asses how good is state-action pair

Notes:

- Values are conditional expectations that update over time
- Value/q-func can be eval from experience (ie sampling).
  - Eg: Monte Carlo methods & Multi-Armed Bandits (MAB).
- Sampling may require storing rewards for all actions on each state, ⇒ hard to maintain.

13.2.1. Optimal Policy π^*

Idea: max rewards sum $\pi^*=\arg\max_\pi\mathbb{E}[G_t|\pi]$ (see 13.1 for G_t)

Optimal value func: $v_{\pi^*}=\max_\pi v_\pi(s)$

Optimal q-func: $q_{\pi^*}(s,a)=\max_\pi\mathbb{E}[G_t|S_t=s,A_t=a,\pi^*]$

13.2.2. Bellman Equation

Idea: optimal policy ⇒ take optimal action (specified by q_*) every state. Solving equation gives π_*.

Equivalent G_t recursion rep: $G_t=R_{t+1}+\gamma\sum_{k=2}^T R_k=R_{t+1}+\gamma G_{t+1}$

Value func: $v_\pi(s)=\mathbb{E}_\pi[G_t|S_t=s]=\mathbb{E}_\pi[R_{t+1}+\gamma G_{t+1}|S_t=s]=\sum_a\pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)[r+\gamma\mathbb{E}_\pi[G_{t+1}|S_{t+1}=s']]=\sum_a\boldsymbol{\pi(a|s)}\sum_{s',r}\boldsymbol{p(s',r|s,a)[r+\gamma v_\pi(s')]}$

Optimal Value Func v_{π^*}: $v_*(s)=\max_a\mathbb{E}[R_{t+1}+\gamma v_*(S_{t+1})|S_t=s,A_t=a]=\max_a\sum_{s',r}p(s',r|s,a)[r+\gamma v_*(s')]$

Optimal Q-Func q_{π^*}: $q_*(s,a)=\mathbb{E}[R_{t+1}+\gamma\max_{a'}q_*(S_{t+1},a')|S_t=s,A_t=a]=\sum_{s',r}p(s',r|s,a)[r+\gamma\max_{a'}q_*(s',a')]$

13.2.3. Finding Optimal Policy: Policy Iteration

Idea: eval & improve policy directly.

13.2.3.1. Policy Evaluation

Idea: asses policy quality by exploring current value/q-func.

Alg: for same policy π:

1. Init (can rand init) optimal value func approximation v_0
2. Use Bellman equation to create updated (= more accurate) value func $v_{k+1}(s)$
3. Repeat until value func converges

13.2.3.2. Policy Improvement

Idea: try diff (than policy) actions & test if improve

Simple Alg: eval changes to only 1 state every time. Def $\pi'(s)$ s.t. $\begin{cases}\pi'(s)\neq\pi(s), & s\\ \pi'(s)=\pi(s), & \text{else}\end{cases}$, then apply Bellman equation & see if $v_{\pi'}(s)\ge v_\pi(s)$. If True, update policy.

Extended Alg: eval changes to multiple states (not just 1): $\pi'(s)=\arg\max_a\mathbb{E}[R_{t+1}+\gamma v_\pi(S_{t+1})|S_t=s,A_t=a]=\arg\max_a\sum_{s',r}p(s',r|s,a)[r+\gamma v_\pi(s')]$

- Note: each time (possibly) change action only for current state & follow original policy for rest of time

13.2.3.3. Policy Iteration

Idea: iteratively apply eval & improve. $\pi_0\xrightarrow{E}v_{\pi_0}\xrightarrow{I}\pi_1\xrightarrow{E}v_{\pi_1}\xrightarrow{I}\pi_2\xrightarrow{E}\cdots\xrightarrow{I}\pi_*\xrightarrow{E}v_*$

- Finite MDP ⇒ finite # of policies ⇒ ensures that alg eventually produce optimal policy

13.2.4. Value Iteration

Problem: policy iteration can be very inefficient since need to eval current policy at all states repeatedly.

Solution: make more efficient by eval each state only once

Alg: eval & improvement steps of policy iteration (see 13.2.3) are eval & truncated (don't proceed beyond single step): $v_{k+1}(s)=\max_a\mathbb{E}[R_{t+1}+\gamma v_k(S_{t+1})|S_t=s,A_t=a]=\max_a\sum_{s',r}p(s',r|s,a)[r+\gamma v_k(s')]$

- Assured convergence

13.2.5. Monte-Carlo (MC) Methods

Idea: model free (no complete knowledge of model so must sample) Other than that, same alg:

1. Pred: sample given policy π
2. Eval: approximate v_π,q_π
3. Improve policy
4. Repeat until convergence

Pros:

- Can learn V,Q directly from env
- No need for env models
- No need to learn about all states

Cons:

- Only works in episodic env.
- Learn from complete episodes (ie no bootstrapping)
- Must wait until episode end to know return

Solution for cons: Temporal Difference (TD) learning (see 13.2.7).

13.2.5.1. MC Prediction

Idea: build value func v.

Episode: finite set of transitions through S while following π

- Cab have multiple occurrences of same state $s_X\in S$.
- Each occurrence is called visit

common MC pred variants:

1. 1st visit MC: estimates returns obtained after 1st visit to s:
   Init list of returns R
   Gen episode E using π
   $\forall s\in E$:
   Add to R the return following 1st occurrence of s
   Mark $V(s)=mean(R(s))$.
2. Every-visit MC: estimates average returns obtained after all visits to s: Change "return following 1st occurrence" to "average return following all appearances"

13.2.5.2. MC Eval

Problem: MC is model free ⇒ can't eval states.

Solution: eval state-action pairs (q-fun)

Idea: eval $q(s,a)$ instead of $v(s)$. Like policy eval, but instead of keeping values only for states, do it for state-action pairs

- Can use 1st/every visit MC

Problem: if π is deterministic ⇒ may never reach some state-action pairs.

Solution (exploring starts): start episode at rand state (ie all states must have $p>0$ to be start point)

13.2.5.3. MC Improvement

Idea: like policy iteration, done greedily, by selecting highest q-function value for each state, based on current estimate (sampling) $\pi(s)=\arg\max_a q(s,a)$

13.2.6. RL On & Off Policy Method

On-policy methods: try to eval/improve policy used to make decision

- More efficient, but still require new samples with each policy change
- Problem: how to explore all possibilities if follow π?
- Solution: slightly deviate to enable exploration

Off-policy methods: try to eval/improve policy other than one used to gen data

- Converge slower, but more powerful & general (can also learn from human experts)
- Problem: may not have data needed to test policy

13.2.6.1. Off-Policy Importance Sampling

Prob of any trajectory: $\{s_t,a_t,s_{t+1},a_{t+1},\dots,s_T\}$ $Pr(s_t,a_t,s_{t+1},a_{t+1},\dots,s_T|s_t,a_{T-1}\sim\pi)=\prod_{k=t}^{T-1}\pi(a_k|s_k)p(s_{k+1}|s_k,a_k)$

Idea: use relative prob of trajectory for target & behavior policies. $p_{t:T-1}=\frac{\prod_{k=t}^{T-1}\pi(a_k,s_k)p(s_{k+1}|s_k,a_k)}{\prod_{k=t}^{T-1}b(a_k,s_k)p(s_{k+1}|s_k,a_k)}=\prod_{k=t}^{T-1}\frac{\pi(a_k s_k)}{b(a_k s_k)}$

Set of returns for time t: $\{G_t\}_{t\in\mathcal{T}(s)}$

Corresponding importance sampling values: $\{p_{t:T(t)-1}\}_{t\in\mathcal{T}(s)}$ for given set of episodes, $\mathcal{T}(s)$ all time stamps state s was visited over multi episodes, $T(t)$ termination time after t given episode

Estimate $v_\pi(s)$: average returns:

$$V(s)=\frac{\sum_{t\in\mathcal{T}(s)}p_{t:T(t)-1}G_t}{|\mathcal{T}(s)|}$$

13.2.6.2. ε-Greedy Algorithms

In RL must balance exploration & exploitation:

- Exploration: experiment with multi actions to better assess rewards
- Exploitation: max rewards by choosing best action

Problem: greedy alg don't explore⇒ may get stuck on sub-optimal actions.

Solution: set all values of Q to very high value, then use MC sampling to drive them down gradually (also requires that that each action was already visited multiple times to ensure gradual degradation)

ε-Greedy Idea: greedy choose best action a^* most times (prob $1-\epsilon$), but with prob ϵ choose rand action.

- Ensures ability to find a^* (thus π^*)
- But, choosing rand actions ⇒ never optimal

13.2.7. Temporal Difference Learning

Idea: sample expected value (MC) then use estimate V instead of actual v_π (bootstrapping)

Overcome MC cons:

- Works in (infinite) envs.
- Can learn & update approximations after each step
- Can learn from incomplete seqs

13.2.7.1. One-Step TD/TD(0)

Idea: update based on 1 step forward

Alg: init $\forall s.\ V(s)=0$, $\forall$episode E:
Init S
$\forall$step $s\in E$:
Get R,s' using action $a=\pi(s)$
$V(s)\mathrel{+}=\alpha[R+\gamma V(s')-V(s)]$
$s=s'$
break if s is last step of episode

13.2.7.2. TD Error δ

Idea: estimation update is based on err: $\delta_t=R_{t+1}+\gamma V(s_{t+1})-V(s_t)$

- Note: err in time t requires info from $t+1$ ⇒ must wait 1 time step to update.
- This holds under if V doesn't change from one step to another

13.2.7.3. On-Policy TD Control: SARSA

Idea: must visit all state-action combinations (like prev on-policy).

Terminal state: $Q(s_T,a)=0$ since updates require next step

Alg:

1. init $Q=0$ $\forall$ terminal state
2. $\forall$episode E:
2.1. Init S
2.2. Get a from s using policy derived from Q (eg ϵ-greedy)
2.3. $\forall$step $s\in E$:
2.3.1. Get R,s' using action a
2.3.2. Get a' from s' using policy derived from Q (eg ϵ-greedy)
2.3.3. $Q(s_t,a_t)\mathrel{+}=\alpha[R_{t+1}+\gamma Q(s_{t+1},a_{t+1})-Q(s_t,a_t)]$
2.3.4. $s=s';a=a'$
2.3.5. break if s is last step of episode

13.2.7.4. Off-Policy TD Control: Q-Learn

Difference no policy, relative to the optimal policy (π and b)

1. init $Q=0$ $\forall$ terminal state
2. $\forall$episode E:
2.1. Init S
2.2. $\forall$step $s\in E$:
2.2.1. Get a from s using policy derived from Q (eg ϵ-greedy)
2.2.2. Get R,s' using action a
2.2.3. $Q(s_t,a_t)\mathrel{+}=\alpha[R_{t+1}+\gamma\max_a Q(s_{t+1},a)-Q(s_t,a_t)]$
2.2.4. $s=s'$
2.2.5. break if s is last step of episode

13.2.7.5. Q-learn vs SARSA

Idea: Q-learn learn q-func relative to optimal policy (greedy approximation) instead of based on current policy

1. Most important diff is Q update rule: SARSA uses Q following ϵ-greedy policy when a' is drawn from it, while Q-learn uses max Q over all possible actions for next step. This makes it look like following greedy policy with $\epsilon=0$ (ie NO exploration).
2. However, when taking action, Q-learning still uses action taken from ϵ-greedy policy (Thus "Choose a ..." is inside repeat loop).
3. Following loop logic in Q-learning a' is still from ϵ-greedy policy.

13.2.7.6. Deep Q-Learning

Idea: instead of calc $q(s,a)$ directly, use DNN to calc approximation $q(s,a)\Rightarrow q(s,a,\theta)=q_*(s,a)=\mathbb{E}[R_{t+1}+\gamma\max_{a'}q_*(S_{t+1},a')|S_t=s,A_t=a]$, ($\theta$ approximation params).

Loss: $L_i(\theta_i)=\mathbb{E}\left[\left(y_i-Q(s,a,\theta_i)\right)^2\right]$ for $y_i=\mathbb{E}[R_{t+1}+\gamma\max_{a'}Q(s,a,\theta_{i-1})|S_t=s,A_t=a]$ score calc with current Q-func

Backprop: $\nabla_{\theta_i}L_i(\theta_i)=\mathbb{E}[R_{t+1}+\gamma\max_{a'}Q(s',a',\theta_{i-1})-Q(s,a,\theta_i)\nabla_{\theta_i}Q(s,a,\theta_i)]$

- Use Bellman equation for iterative updates of policy approximation

13.3. Policy Gradients

Idea: model policy directly instead of value funcs (can use value funcs) & improve performance using grad Ascent $\theta_{t+1}=\theta_t+\alpha\widehat{\nabla J}(\theta_t)$ for performance grad estimate $\widehat{\nabla J}(\theta_t)$

- Use grad to update policy params (better params⇒ better actions ⇒ better performance)

Policy output:

- Discrete⇒Probs vec for each action
- Continuous⇒Gaussian mean & covariance

Pros:

- action probs change smoothly, while greedy may dramatic changes
- Model probs of taking action
- Learn appropriate balance of exploration/exploitation
- Naturally handle continuous (infinite)
- (sometimes) easier to rep parameterized policy than a value-function

Main disadvantage:

- Require lots of sampling & often slow to converge

Solution: actor-critic algorithms

13.3.1. REINFORCE Algorithm

Policy grad theorem: $\nabla J(\theta)\propto\sum_s\mu(s)\sum_a q_\pi(s,a)\nabla_\theta\pi(a|s,\theta)\overset{(1)}{=}$

To enable MC sampling need to transform formula s.t. it uses samples A_t,S_t

For a definition of G_t, see "goals and rewards" in lecture 1

$$\mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a)\nabla_\theta\pi(a|S_t,\theta)\right]\overset{(2)}{=}\mathbb{E}_\pi\left[\sum_a\pi(a|S_t,\theta)q_\pi(S_t,a)\frac{\nabla_\theta\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}\right]\overset{(3)}{=}\mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla_\theta\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right]\overset{(4)}{=}\mathbb{E}_\pi\left[G_t\boldsymbol{\frac{\nabla_\theta\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}}\right]$$

for $\propto$ is "proportional to", $\mu(s)$ is average # of times state s was visited in trajectory.

1. Replaced sum over states with $\propto$ value of sample $S_t\sim\pi$
2. Multiply & divide by $\pi(a|S_t,\theta)$ prob to select action a in π & state S_t
3. Replace a with $A_t\sim\pi$
4. Since $\mathbb{E}_\pi[G_t|S_t,A_t]=q_\pi(S_t,A_t)$ (see 13.1 for G_t)

- Note: sample grad is only $\propto$ since can use LR α to correct diffs in $\propto$

Update policy:

$$\theta_{t+1}=\theta_t+\alpha G_t\frac{\nabla_\theta\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

- Update based on rewards received from that point on & chance of taking that action (grad of prob divided by prob)
- Param vec values are updated $\propto$ to reward & in reverse $\propto$ to prob (ie likelier actions ⇒ smaller effect)
- MC based ⇒ all updates are made after trajectory completion
- Only works in episodic (finite) setting

Alg: for differentiable policy parameterization $\pi(a|s,\theta)$
Repeat forever:
Gen episode $E=\{S_0,A_0,R_0,\dots,S_{T-1},A_{T-1},R_{T-1}\}$, following $\pi(\cdot|\cdot,\theta)$
$\forall$ step of E $t=0,\dots,T-1$:
G_t = return from step t
$\delta=G_t-\hat{v}(S_t,w)$
$w\mathrel{+}=\alpha^w\gamma^t\delta\nabla_w\hat{v}(S_t,w)$
$\theta\mathrel{+}=\alpha\gamma^t\delta\nabla_\theta\ln\pi(A_t|S_t,\theta)$
Red added for baseline (see 13.3.1.1)

13.3.1.1. REINFORCE with Baseline

Problem: sampling use⇒ alg might fluctuate & converge slow (ie high var)

Solution: include baseline (value that reduces update size) $\theta_{t+1}=\theta_t+\alpha(G_t-b(S_t))\frac{\nabla_\theta\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$

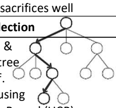
- Most important: must not vary with a, else baseline will be correlated with what is optimize
- Logical choice for $b(S_t)$: state-value func $\hat{v}(S_t,w)$
  - w is calc by method other than policy grads
  - Can also be updated by sampling
  - Rewards actions that outperform "average" for state
- Converge guaranteed (due to grad ascent), but process is slow (like all MC approaches).
- Hard to implement for non-episodic/infinite problems.

13.4. Monte Carlo Tree Search (MCTS)

Idea: combine MC sampling & tree search. Search tree can rep states & actions transit one state to another. Action choice at any state is governed by tree policy π. During train, policy takes to account both exploration & exploitation. Once reach leaf, expand tree via sampling, update policy based on outcome & start over.

- No need for domain-specific knowledge
- Can interrupt at any time & provides best answer it has at moment
- Enables fixed time/budget train
- Hard to model sacrifices well

13.4.1. MCTS: Selection



Idea: start at root & navigate current tree until arrive to leaf. Navigation done using Upper Confidence Bound (UCB):

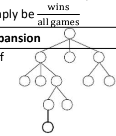
$$\pi_{UCB}(s)=\arg\max_a\underbrace{Q(s,a)}_{\text{Exploitation}}+c\underbrace{\sqrt{\frac{\ln n(s)}{n(s,a)}}}_{\text{Exploration}}$$

For $n(s)$ # of times state s was visited, $n(s,a)$ # of times action a was taken in state s, c param that calibrates exploration/exploitation balance (exploration usually have larger weight at start & then reduced over time)

- $Q(s,a)$ can simply be $\frac{\text{wins}}{\text{all games}}$

13.4.2. MCTS: Expansion

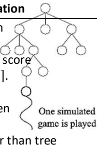
Once reach to leaf either:



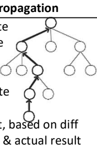
- Game end (go straight to backprop)
- Init all possible child nodes (weights 0) using all legal actions. Use rollout policy to select 1 action & add it to graph

13.4.3. MCTS: Simulation

Run single game from expanded node until termination, observe score at game end $\in[-1,1]$. Actions during simulations are chosen either rand or using rollout policy (simpler than tree policy). Can run >1 simulation at each iteration (easy to parallelize)



13.4.4. MCTS: Backpropagation



With outcome, update $Q(s,a)$ for each node that participated in game:

- Using simple update win/loss ratio
- Using grad descent, based on diff between expected & actual result

13.4.5. Combine MCTS with Policy

Software policy def using hand-crafted features. Once expanded node has had enough simulations, policy used to determine if it should remain in tree

- Slows tree growth while focusing attention on high-prob moves

13.4.6. Combine MCTS & Value Func

Q-func: $Q_{RLGO}(s,a)=\sigma(\phi(s,a)^T\theta)$ for ϕ binary features & θ weights, σ map to estimated prob to win

For sampling process (instead of rand policy) can use:

- ϵ-greedy policy
- Greedy policy with noisy value func $\pi_\sigma(s,a)=1\Leftrightarrow a=\arg\max_{a'}\left(Q_{RLGO}(s,a')+\eta(s,a')\right)$
  - Noise with mean 0, can cause non-optimal actions to receive higher values than optimal
- SoftMax dist with temperature τ: $\pi_\tau(s,a)=\frac{\exp Q_{RLGO}(s,a)/\tau}{\sum_{a'}\exp Q_{RLGO}(s,a')/\tau}$

13.5. AlphaGo

Idea: use 3 nets: supervised learn policy, RL policy, RL value, and use lin SoftMax model as rollout policy.

13.6. שאלות ממבחנים

ל-beam search בבעיות תרגום, שלא כמו ל-greedy search, אין העדפה מובנית לרצפים קצרים יותר. **לא נכון**

ב-beam search עם פרמטר B=3, מספר המועמדים באיטרציה i נקבע ע"פ הנוסחא $\log(B^i-1)$. **לאנכון**

Generative stochastic networks מבצעות תפקיד דומה לזה של רכיב ה-Generator ב-GANs. **נכון**

Generative stochastic networks יכולות להיחשב כסוג של denoising autoencoder. **נכון**

policy המבוססת על תצפיות תמיד תהיה טובה לפחות כמו מדיניות מבוססת מצבים. **לא נכון**

ב-RL עם אופק אינסופי ו- stationary state distribution, ניתן להתעלם מכל מצבי העבר כאשר מחשבים את ה-expected rewards. **נכון**

ב-GANs, ה-Generator מקבל את הקלט z. קלט זה הוא ה-latent factors שנדגמו באופן ראשוני מהנתונים האמיתיים. **לא נכון**

בעיית ה-mode collapse ב-GANs יכולה להיפתר ע"י שינוי פונקציית ה-loss של ה-Generator כך שתוציא ערכי גראדיינט גדולים יותר כפלט. **לא נכון**

מטרתה של רגולריזציה הינה להקטין את ה-variance אך לא את ה-bias. **נכון**

Adversarial training יכול להקטין הן את ה-bias והן את ה-variance. **נכון**

כאשר אנו משתמשים ב-Q-function או ב- value function כדי לבחור את הפעולות בכל state, קיימת האפשרות לא להגדיר במפורש policy לפיה נפעל. **נכון**

Q-function מתאימה רק ל-stochastic system **לא נכון**

Neural style transfer. בעת חישוב רכיב ה-cost משתמשים לרוב ברשת pre-trained ומחשבים את הפרש הערכים של ה-softmax עבור כל אחד מה-classes. **לא נכון**

בעת חישוב רכיב ה-style, משתמשים במטריצת המחשבת את הקורלציה בין ההפעלות של הפילטרים השונים ב-hidden layer נתונה. **נכון**

בעת השימוש ב-autoencoders לצורך תרגום, אנו מקבלים משפט בשפה אחת ומוציאים את התרגום באופן סדרתי (מילה-מילה). אסטרטגיה אחת היא לבחור את המילה בעלת ההסתברות הגבוהה ביותר בכל איטרציה (greedy search). **אסטרטגיה זו תוביל לכך שבכל נקודה יבחרו המילים או הביטויים הנפוצים ביותר, דבר שיכול להוביל לכך שתרגומים מדויקים יותר עם ביטויים שכיחים פחות לא יבחנו כלל.**

האם הוספת L2 norm regularization יכולה לסייע בפתרון בעיית ה-vanishing gradient. **לא, סוג זה של רגולציה מקטין את ערכי המשקולות ולכן לא יסייע בפתרון בעיית הvanishing gradient.**

הסבר את העיקרון העומד מאחורי השימוש באלגוריתם momentum. **מתבצע ממוצע ממושקל של הגרדיאנטים ע"פ מס' איטרציות. המיצוע מנטרל תנודות בכיוונים מנוגדים ומאפשר התכנסות מהירה יותר בכיוון "הנכון".**

ציינו דוגמה אחת לבעיה בה bi-directional RNN יהיה עדיף על "Vanilla RNN" ודוגמא אחת בה Bi-directional RNN אינו ישים. **חיובי – תיוג זהויות. שלילי – השלמת משפטים**

האם max(0.2z, 2z) תחליף החactivation. **כן, leaky relu**

מהם היתרונות של השימוש ב-vectorization בהכפלת מטריצות- **ניתן להאיץ את תהליך האימון ע"י ניתוח batch שלם בבת אחת ללא שימוש בלולאות**

ResNet היא ארכיטקטורת CNN בעלת skip connections המקשרות את שכבה i לשכבה 2+i. הארכיטקטורה הוכחה כיעילה בהתמודדות עם בעיית ה-vanishing gradient בגלל (הנח שימוש ב- ReLU and L2 norm):**שכבות שתרומתן לפתרון הבעיה נמוכה, המשקולות שלהן יעודכנו לערכים הקרובים ל-0. ה-skip-connections מאפשרים "לדלג" מעל שיכבה זו ובכך למנוע את התאפסות הגראדיינט**

אלגוריתם ה-sliding window הבסיסי לזיהוי עצמים משתמש במספר שכבות convolution ו-maxpool, ולאחר מכן בשכבה אחת או יותר מסוג fully-connected. האלגוריתם אפקטיבי אך איטי. כיצד נוכל לשפר את יעילותו- **להחליף את השכבות ה-fully connected ב-1x1 convolutions, ע"י שמירת ערכי הביניים שחושבו על ידי פילטרים ב-sliding windows חופפים ניתן להימנע מחישוב חוזר של ערכים**

אתם מאמנים רשת נוירונים עמוקה. לאחר מספר epochs ה-loss function מפסיקה לרדת. מה תעשו - **הקטנת ה-learning rate**

הסבר כיצד השימוש ב-teacher forcing יכול להאיץ את האימון של auto regressive models. ניתן להשתמש ברשת PixelCNN כדוגמה. **במודל מסוג זה, כל חיזוי באלה שקדמו לו. ע"י הזנת הערכים האמיתיים אנחנו מסוגלים למקבל את תהליך האימון.**

על מנת למדל תלויות ארוכות טווח ב-GRU - **שער ה-reset צריך להיות מאוד לא-פעיל (עם פלט קרוב ל-0), שער ה-update צריך להיות פעיל מאוד (עם פלט קרוב ל-1)**

בעת השימוש ב-autoencoders, ה-hidden layer היא בעלת ממדיות נמוכה יותר משל הקלט (ממדיות גבוהה יותר מחייבת שימוש רק בחלק מהנוירונים לכל קלט). מדוע דרישה זו חשובה (רמז: לא מדובר ב-robustness). **אם הhidden layer היא בעלת ממדיות זהה או גבוהה מזו של הקלט, היא פשוט תחזיר את הקלט המקורי**

ב-variational learning אנו מסתמכים על ה-variational bound. אילו מההצהרות הבאות נכונה: - **ה-variational bound מיועד להגביל את ההבדל בין ה-modeled distribution להתפלגות האמיתית**

הניחו שקיבלתם dataset של סרטוני וידאו שצולמו ממצלמות שהורכבו על מכוניות בזמן נהיגה. מטרתכם היא להשתמש ב-deep reinforcement learning כדי לבנות מודל לנהיגה אוטונומית. אילו מההצהרות הבאות נכונה: **עלינו להשתמש באלגוריתם Q-learning מכיוון שזה אלגוריתם off-policy**

בחרו את התשובה הנכונה ביותר עבור auto regressive models - **הם ניתנים להפעלה גם בדומיינים בהם אין לנתונים סדר טבעי, אך יש לקבוע סדר שרירותי כלשהו, הם ניתנים להפעלה רק בדומיינים בהם ניתן להגדיר explicit density function**

אילו מההצהרות הבאות נכונות עבור ארכיטקטורת GAN - **תרחיש בו הgenerator- משפר את ביצועיו אך אלה של הdiscriminator- נשארים ללא שינוי הוא אפשרי**

היותה של רשת PixelCNN אלגוריתם regressive auto מתבטאת ע"י - **הן המעבר הסדרתי על כל פיקסל והן ע"י ההפעלה הסדרתית על כל צבע (בייצוג)RGB בפיקסל.**

אילו מהפעולות הבאות יכולה לסייע בסיווג נכון של אובייקטים חופפים בתמונות - **הכפלת גודל וקטור הפלט במספר הישויות אותן רוצים לסווג במקביל (למשל במקום 8 כניסות עבור סיווג ישות אחת, עבור שתי ישויות יהיו 16 כניסות).**

הניחו כי אתם מאמנים רשת נוירונים תוך שימוש ב-training וב-validation. לאחר מספר epochs אתם רואים כי ה-training loss נמוך בהרבה מאשר ה-validation loss. מה מהדברים הבאים יכול לסייע בפתרון הבעיה: **להשתמש ברשת עם פחות שכבות, להשתמש ב-L2.**

מה מההצהרות הבאות נכון לגבי השימוש ב?dropout- - **השימוש בdropout- מקטין את הסיכוי לoverfitting- על סט האימון, השימוש בdropout- יכול לגרום להקטנת המשקולות השונים ברשת, על ידי "העלמת" נוירונים אקראיים ברשת אנחנו מקטינים את הסיכון שהמודל יסתמך רק על חלק מהמידע המיוצג בנתונים**

ניתן להשתמש בGAN- לצורך ביצוע de-noising. בחרו את התשובה הנכונה: - **יש "להרעיש" את הקלט המקורי (לדוגמה באמצעות)dropout ע"מ ללמד את הרשת להשלים תמונות, בסופו של תהליך האימון, הרכיב היחיד בו נשתמש הוא הgenerator-**

כאשר משתמשים בארכיטקטורת encoder-decoder עם Models Attention לצורך translation machine - **בשלב ה,decoder- מודל הattention- מאפשר להתמקד במספר אזורים של הקלט בו זמנית**