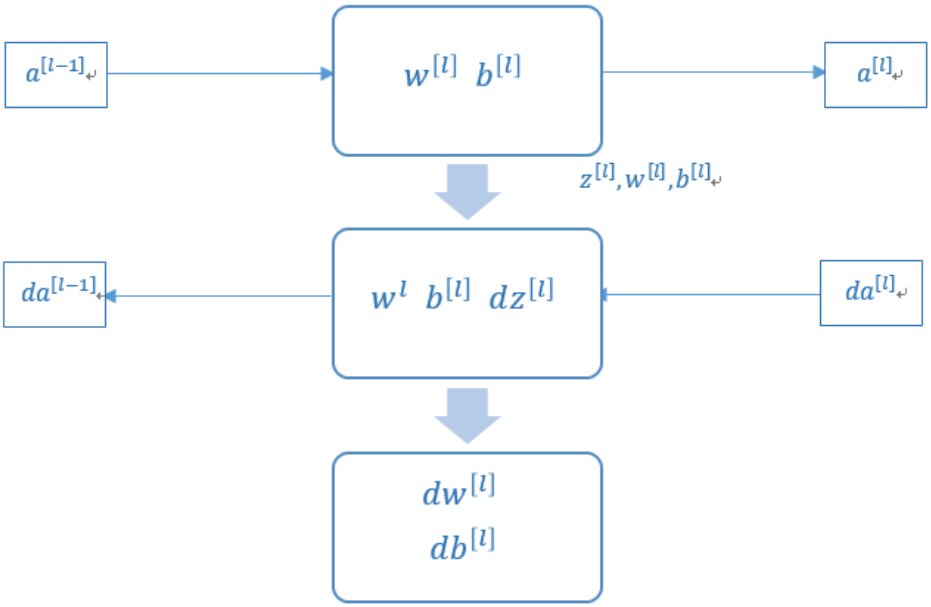
Course 1:

How to train a neural network?



$$dZ^{[l]} = dA^{[l]} * (g^{[l]})'Z^{[l]}$$

$$dw^{[l]} = \frac{1}{m} dZ^{[l]} \cdot (A^{[l-1]})^T$$

$$db^{[l]} = \frac{1}{m} dZ^{[l]}$$ (Using np.sum($dZ^{[l]}$, axis=1, keepdims=True))

$$dA^{[l-1]} = \left(w^{[l]}\right)^T \cdot dZ^{[l]}$$

Loss function: $L(\hat{y}, y) = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})]$

Cost function:

$$J = -\frac{1}{m}\sum_{i=1}^{m} (y^{(i)}\log(a^{[L](i)}) + (1-y^{(i)})\log(1-a^{[L](i)}) = \frac{1}{m}\sum L(\hat{y}, y)$$

For final layer: $dA^{[L]} = -\frac{y^i}{da^i} + \frac{1-y^i}{1-a^i}$.

Why we choose this loss function:

if y=1, p(y|x)= $\hat{y}$; if y=0, p(y|x)= $1-\hat{y}$

so p(y|x)= $\hat{y}^y \cdot (1-\hat{y})^{1-y}$, $\log(p) = -L(\hat{y}, y)$. We want L becomes lower when p is higher, so we use $-L(\hat{y}, y)$.

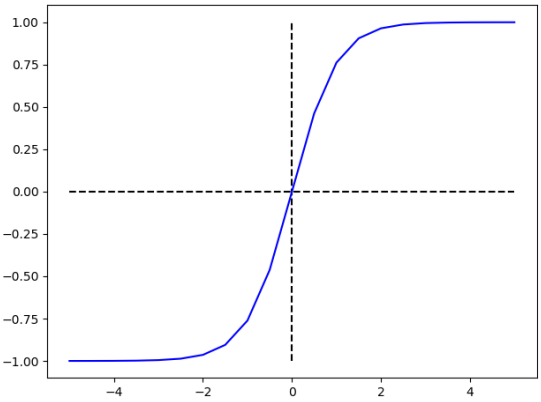
Course 2:

Regularization: often use L2 regularization.

①Logistic: $J = \frac{1}{m}\sum L(\hat{y}, y) + \frac{\lambda}{2m}||w||^2$

②NN: $J = \frac{1}{m}\sum L(\hat{y}, y) + \frac{\lambda}{2m}\sum ||w^{[l]}||^2_F$

Interpretation: $dw^{[l]} \equiv \frac{\partial J}{\partial w^{[l]}} = [backprop] + \frac{\lambda}{m} w^{[l]}$, $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$, which becomes smaller. And $\lambda \uparrow$, $w \downarrow$, $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ so $z \downarrow$.



From the pic we can see that if $|z|$ is small, it's in a linear range. $g(z)$ is simple.
Other ways to regularization: drop-out.

Before implementing the NN, remember to normalization.
Weight initialization:

1. The weights of $w^{[l]}$ should be initialized randomly to break the symmetry. But it's OK to initialize all $b^{[l]} = 0$

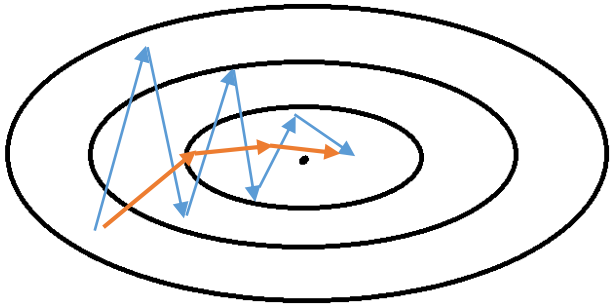
2. For deep models, if w too big, the $\hat{y}$ will explode. And if w too small, the $\hat{y}$ will decay fast. So we use

$w^{[l]} = \text{np.random.randn(/shape/)} * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$ for ReLU activate function.

And using $\sqrt{\frac{1}{n^{[l-1]}}}$ for tanh.

Optimization algorithms
1. Mini-batch gradient descent: choose a subset of inputs if the number is huge
   Usually use 64-512
2. Exponentially weighted averages
3. Gradient descent with momentum
   Momentum takes into account the past gradients to smooth out the update.
   On iteration t:
   compute dW, db on the current mini-batch
   $v_{dW} = \beta v_{dW} + (1-\beta) dW$ # β is often choosed at 0.9
   $v_{db} = \beta v_{db} + (1-\beta) db$
   $W = W - \alpha v_{dW}, b = b - \alpha db$

It can make the vertical direction averaged and keep the horizontal fast.



4. RMSprop

On iteration t:

compute dW, db on the current mini-batch

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2$ (dW is small)

$S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$ (db is large)

$$w = w - \frac{\alpha dw}{\sqrt{S_{dW}}}$$

$$b = b - \frac{\alpha db}{\sqrt{S_{db}}}$$

(we consider it in the $b - w$ space)

5. Adam optimization algorithm (combine the two above)

Initialize $v_{dW} = S_{dW} = v_{db} = S_{db} = 0$.

On iteration t:

compute dW, db on the current mini-batch

$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$

$v_{db} = \beta_1 v_{db} + (1-\beta_1)db$

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2$

$S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$

$$v_{dW}^{corr} = \frac{v_{dW}}{1-\beta_1^t}, v_{db}^{corr} = \frac{v_{db}}{1-\beta_1^t}$$

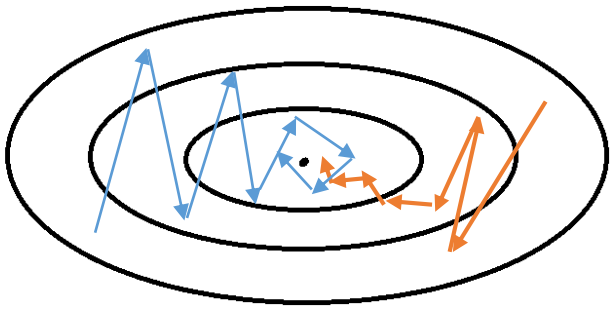
$$S_{dW}^{corr} = \frac{S_{dW}}{1-\beta_2^t}, S_{db}^{corr} = \frac{S_{db}}{1-\beta_2^t}$$

$$w = w - \frac{\alpha v_{dW}^{corr}}{\sqrt{S_{dW}^{corr}}+\varepsilon}, b = b - \frac{\alpha v_{db}^{corr}}{\sqrt{S_{db}^{corr}}+\varepsilon}$$

(add ε to make sure that it's not zero)

6. Learning rate decay

Take smaller steps after some iterations because it may oscillate at the optimal point.



Then it's more likely to converge in a small range.

Tuning process

The most important: α. Also mini-batch size, hidden units

No grid search, using random sampling of parameters.

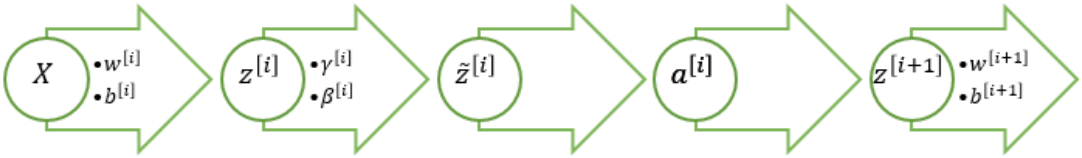
Batch normalization

Normalize each layer's output $z^{[l]}$

$$z^i_{norm} = \frac{z^i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^i = \gamma z^i_{norm} + \beta$$

γ and β are learnable parameters, which help to get the range we want. (not just 0-1)



Note:

1. During the batch normalization, $b^{[l]}$ will be useless because $z^i = z^i - \mu$ and $b^{[l]}$ is constant. So we use gradient descent to calculate β, γ and w.
2. When doing the test job, we can use σ and μ from training mini-batches.

Multi-class classification:

softmax activation function

$t = e^{z[L]}$ # n_classes classification

$a^{[L]} = \frac{e^{z[L]}}{||t||}$ # normalization

$$\hat{L}(y, \hat{y}) = -\sum_{j=1}^{n_c} y_j \log \hat{y}_j = -\log \hat{y}_{true}$$

$y = (0,0, \dots, 1, 0 \dots)^T$ # $y_j = \delta_{j,true}$ $\hat{y}_{true} = (0,0, \dots, a, 0 \dots)^T$

Backprop: $dZ^{[L]} = \hat{y} - y$

Tensorflow
Structure:
(1) create tensors (variables) / placeholders # eg. x=tf.placeholder(tf.float32, name='x')
(2) write operations between tensors
(3) Initialize # init=tf.global_variables_initializer( )
(4) Create a session # with tf.Session( ) as session:
(5) Run # seesion.run(...)

Build a Neural Network for multi-class classification:
(1) preprocessing & convert to one-hot vector
(2) create placeholders (check the shape)
(3) initialize parameters

eg.

```
W1 = tf.get_variable("W1", [#size],
                     initializer = tf.contrib.layers.xavier_initializer(seed = 1))
b1 = tf.get_variable("b1", [#size], initializer = tf.zeros_initializer( ))
```

(4) forward propagation (for each layer calculate $z^{[i]}$ and $a^{[i]}$

(5) compute cost

eg.

```
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
                    (logits = ..., labels = ...) )
```

make sure that logits/labels = tf.transpose(...) to fit the Tensorflow.
(6) Back propagation

```
optimizer = tf.train.GradientDescentOptimizer
                (learning_rate = learning_rate).minimize(cost)
_ , minibatch_cost = sess.run([optimizer, cost], feed_dict=
                {X: minibatch_X, Y: minibatch_Y})
```

Course 3:

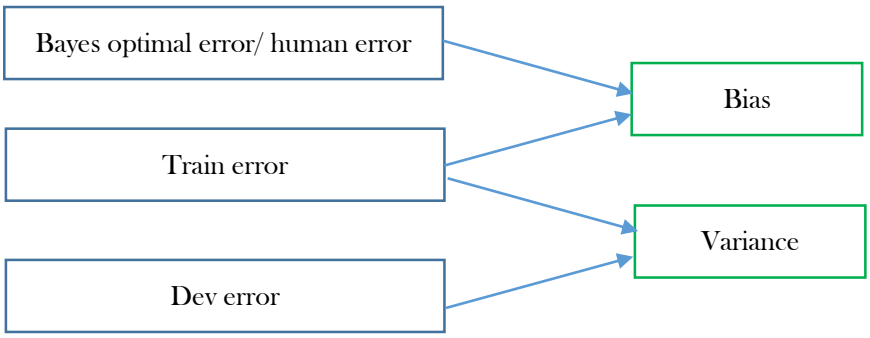
Error metric

$$\text{Error} = \frac{1}{\sum w^i} \cdot \sum w^i \cdot I(y^i_{pred} \neq y^i)$$

eg.

$w^i = \begin{cases} 1 & \text{if } x^i \text{ is normal} \\ 100 & \text{if } x^i \text{ is abnormal} \end{cases}$, then we can give penalty to incorrect/unwanted result.

Bias and variance



Which should be the direction depends on which is bigger. (bias and variance)

Data mismatch: can make artificial synthetic training set (be cautious about the overfitting)

Transfer learning: conditions (1) Task A and B have same input X; (2) more data for task A.

Multi-task learning

x -> ... ->  -> y

$$J = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}^i_j, y^i_j)$$

Compare with softmax: multi-task vs. multi-class

Conditions: (1) data for each task is similar (2) benefit from sharing lower-level fetures

eg. self-driving car, visual detection

Course 4: Convolutional Neural Network

Edge detection

image* filter -> outcome matrix

eg.

0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

*

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

vertical filter (f×f)

0	-30	-30	0
0	-30	-30	0
0	-30	-30	0
0	-30	-30	0

python: conv_forward Tensorflow: tf.nn.conv2d

dark->light /light-> dark: using absolute value
Use back-prop to train the best filter for specific example.

Drawbacks: (1) image shrink (2) edge/corner pixels information less used
Improvement: padding

0	0	0	0	0	0
0					0
0			.....		0
0					0
0					0
0	0	0	0	0	0

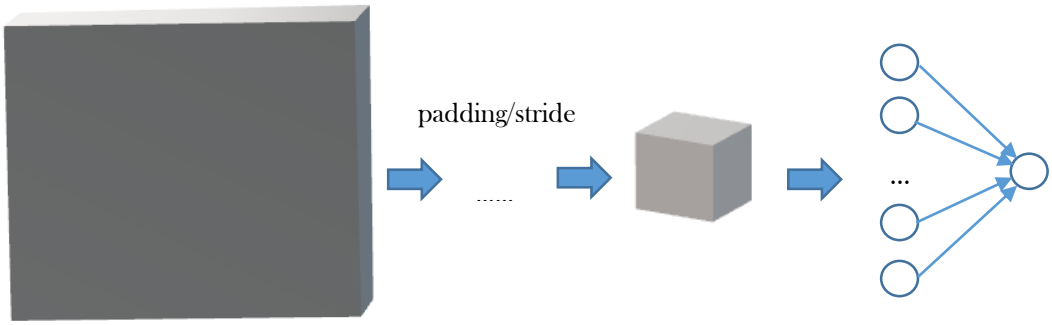
Usually padding(p) $= \frac{f-1}{2}$, f is odd.

if p=0, valid convolution; if p= $\frac{f-1}{2}$, same convolution.

Strided convolution: stride(s)

$$\text{output size} = \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

Convolution over volumes:



_×_×3 _×_×40 flatten

Parameters: (1) filter size $f^{[l]}$ (2) number of channels $n_c^{[l]}$

(3) weights $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ (4) bias $1 \times 1 \times 1 \times n_c^{[l]}$

Type of Layers: (1) convolution (2) pooling (3) fully connected
Pooling layers: eg. max pooling, average pooling
fixed f and s. usually padding=0 (no parameter to learn)

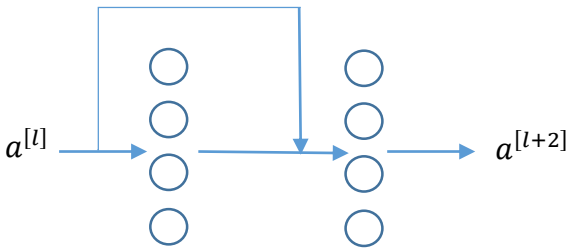
structure of convolution NN:
ConV -> Pool -> ConV -> Pool ->... -> FC -> FC -> SoftMax/ Logistic
Each "ConV+Pool" is one layer.

Advantages of Convolution NN:
(1) parameter sharing. use the same filter
(2) sparsity of connection: each output just depends on a small number of inputs.

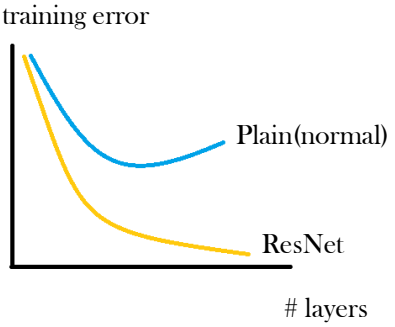
Some classical networks (with references):
LeNet-5, AlexNet, VGG

Advanced network:
ResNets

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



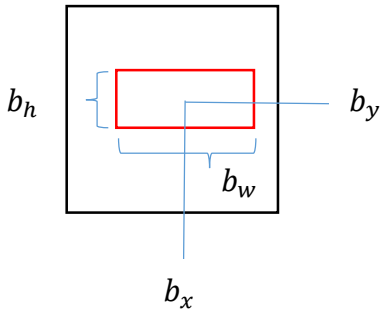
every 2 Conv layers.



Object detection:

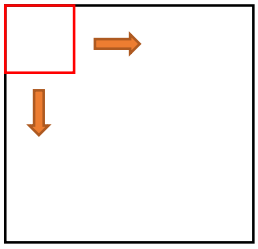
$$y = \begin{pmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ \dots \end{pmatrix}$$

p_c=0 if no object detected and =1 if detected.



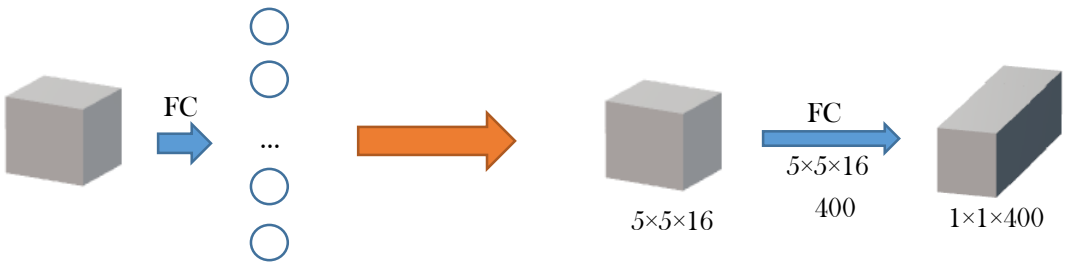
c_i represents classes. (eg. cars, buildings, pedestrians ...)

Sliding windows detection:



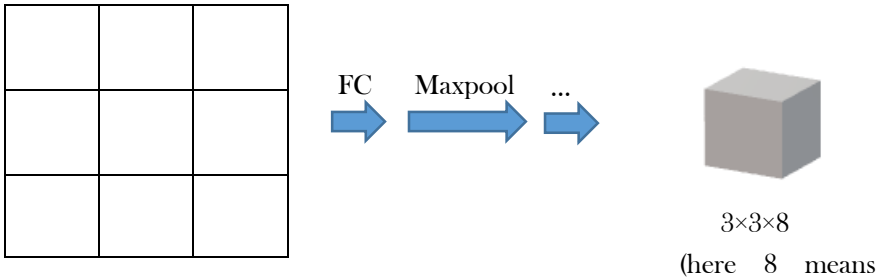
low efficiency by sliding. So we use convolutional layers.

Turn FC layer into convolutional layer:
eg.



Use linear function to these 400 filters, so FC layer equals to convolutional layer.

application: YOLO algorithm

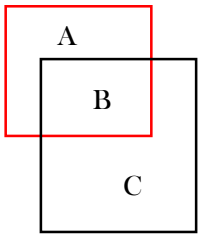


100×100×3

3×3×8
(here 8 means dimension of $\hat{y}$)

divide into 3×3 grid cell. For each cell, we finally get a 1×1 FC layer to represent the result.

evaluation: IoU (Intersection of Union)



$$\text{IoU} = \frac{\text{B}}{\text{A} + \text{B} + \text{C}}$$

Face recognition

similarity function:

if d(img1, img2) <τ

"same person"

$x_1 \to f(x_1) \quad x_2 \to f(x_2)$

$\text{d}(x_1, x_2) = \left||f(x_1) - f(x_2)|\right|^2$ ("Deepface")

Triplet loss function:

P: positive N: negative A: anchor

target: $\left||f(A) - f(P)|\right|^2 + \alpha \leq \left||f(A) - f(N)|\right|^2$, α is the margin.

$$\text{L} = \max\left(\left||f(A) - f(P)|\right|^2 + \alpha - \left||f(A) - f(N)|\right|^2, 0\right)$$

$$\text{J} = \sum L_i$$

Training set should contain pics from same persons. Use GD to train.

Other similarity functions:

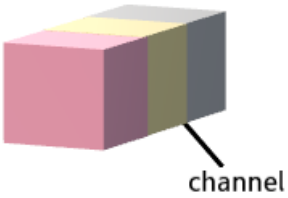
(1) $\hat{\text{y}} = \sigma\left[\text{w}^{\text{T}}\left(f(x_1) - f(x_2)\right) + b\right]$

(2) via supervised learning. Input (img1, img2), $\hat{\text{y}} = \begin{cases} 1 & \text{if same person} \\ 0 & \text{if not same person} \end{cases}$.

Style transfer:
content+style -> generated

Loss function $J(G) = \alpha J_c(C, G) + \beta J_s(S, G)$
Initialize G randomly and then use GD.

$J_c(C, G) = \frac{1}{2}\left|\left|a^{[l](C)} - a^{[l](G)}\right|\right|^2$ use hidden layer l to compute.



style: correlation between activations across channels.
Style matrix:

$a^{[l]}_{ijk}$= activation at (i, j, k), $G^{[l]}_{nw}$.shape= $n^{[l]}_C \times n^{[l]}_C$

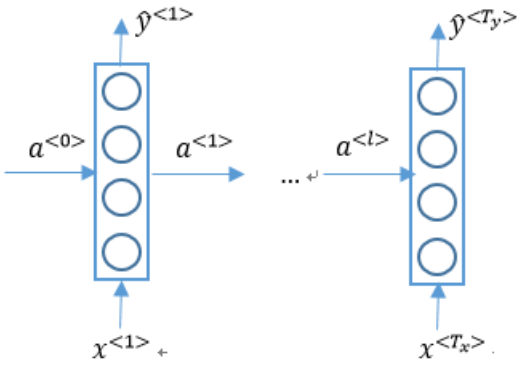
$$G^{[l](S)}_{kk'} = \sum_i^{n_H}\sum_j^{n_W} a^{[l](S)}_{ijk} a^{[l](S)}_{ijk'}$$

the same for $G^{[l](G)}_{kk'}$(generated image)

$$J^{[l]}_s(S, G) = c\left|\left|G^{[l](S)} - G^{[l](G)}\right|\right|_F^2 = \frac{1}{\left(2n^{[l]}_H n^{[l]}_W n^{[l]}_C\right)^2}\sum_k\sum_{k'}\left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)$$

can use $J_s = \sum_l \lambda^{[l]} J^{[l]}_s$ to give more weight on deeper layer.

Course 5 Recurrent Neural network
sequence data
eg. Names detection: using one-hot (with dictionary) to represent words



$a^{<l>} = g(w_{aa}a^{<l-1>} + w_{ax}x^{<l>} + b_a)$ tanh/Relu
$\hat{y}^{<l>} = g'(w_{ya}a^{<l>} + b_y)$ sigmoid

can be simplified: $a^{<l>} = g(w_a[a^{<l-1>}, x^{<l>}] + b_a)$
$[a^{<l-1>}, x^{<l>}] = \begin{pmatrix} a^{<l-1>} \\ x^{<l>} \end{pmatrix}, w_a = (w_{aa}, w_{ax})$

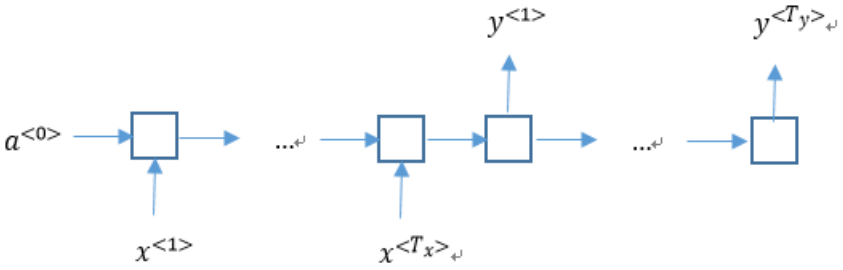
Loss function: cross-entropy “Back-prop through time”

$$L^{<t>}\left(y^{<t>},\hat{y}^{<t>}\right) = -y^{<t>}\log\hat{y}^{<t>} - (1-y^{<t>})\log\left(1-\hat{y}^{<t>}\right)$$

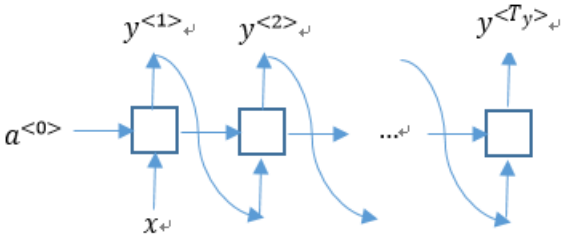
$$\hat{L}(y,\hat{y}) = \sum_{t=1}^{T_y} L^{<t>}(y,\hat{y})$$

Architecture of RNN

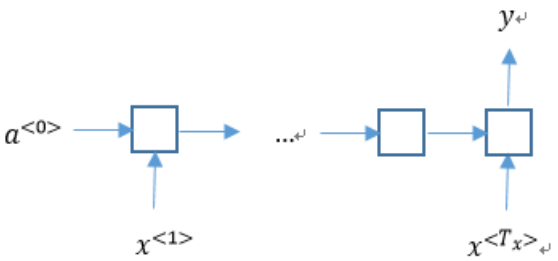
many to many: ($T_x \neq T_y$)



one to many:



many to one:



Vanishing gradient with RNN:

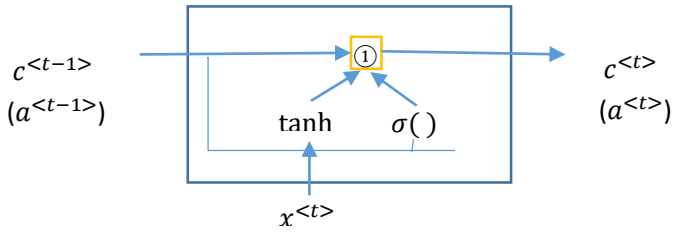
dependency between words in long sentences. hard to back-prop to early word.

GRU: gated recurrent unit

c: memory cell

$$
\begin{cases}
c = a^{<t>} \\
\tilde{c}^{<t>} = \tanh(w_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_r = \sigma(w_r[c^{<t-1>}, x^{<t>}] + b_r) \\
\Gamma_u = \sigma(w_u[c^{<t-1>}, x^{<t>}] + b_u) \\
c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \quad ①
\end{cases}
$$

Γ_u: 0/1 gate. if $\Gamma_u = 0$, $c^{<t>} = c^{<t-1>}$. memory maintained.

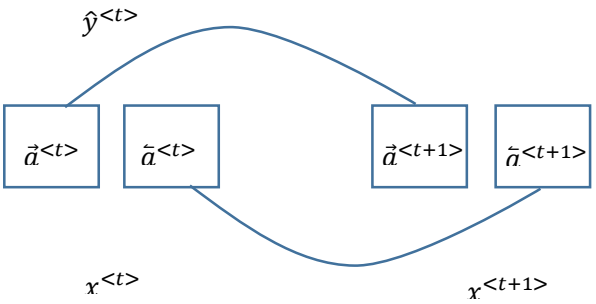


LSTM: long short term memory

$$
\begin{cases}
\tilde{c}^{<t>} = \tanh(w_c[a^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u = \sigma(w_u[a^{<t-1>}, x^{<t>}] + b_u) \\
\Gamma_f = \sigma(w_f[a^{<t-1>}, x^{<t>}] + b_f) \\
\Gamma_o = \sigma(w_o[a^{<t-1>}, x^{<t>}] + b_o) \\
c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \\
a^{<t>} = \Gamma_o * \tanh c^{<t>}
\end{cases}
$$

Γ_u: update Γ_f: forget Γ_o: output

Bidirectional RNN (BRNN)



Word embedding

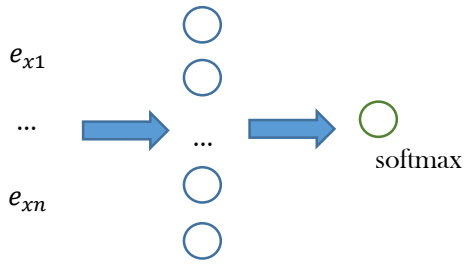
eg.

	"man"	"women"	"apple"	...
Gender	-1	+1	0.05	
Age	0.03	0.08	0.02	
food				
...				

Each word is represented by one-hot vector. e_i is the column of word(i) in the dictionary.
(1) can use t-SNE to group these words.
(2) make analogies: $e_i - e_j$ with $e_{i'} - e_{j'}$, use cos-similarity between $e_{j'}$ and $e_j - e_i + e_{i'}$.

$EO_i = e_i$, E is the embedding matrix, O_i is the one-hot vector.

Word prediction:



context: last few words/ nearby 1 word

softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$ $L = -\sum_{i=1}^{s} y_i^T \log \hat{y}_i$ y_i: one-hot encoding

too much computation!

Negative sampling:
eg.

Context(c)	Word(t)	Target(y)
orange	juice	1
orange	king	0
orange	man	0
...	...	...

k random negative samples and k+1 in total.
$P(y = 1|c, t) = \sigma(\theta_t^T e_c)$

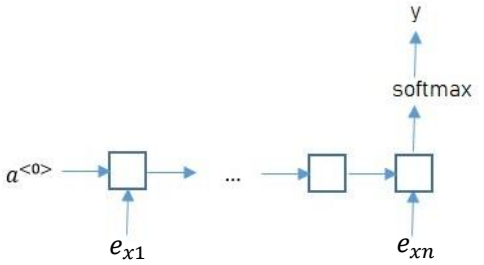
GloVe: global vectors for word representation

$$\min \sum_{i=1}^{10000} \sum_{j=1}^{10000} f(x_{ij})(\theta_i^T e_j + b_i - b_i' - \log x_{ij})^2$$

$f(x_{ij}) = 0$ if $x_{ij} = 0$. to make sure that $x_{ij} \log x_{ij} = 0$.

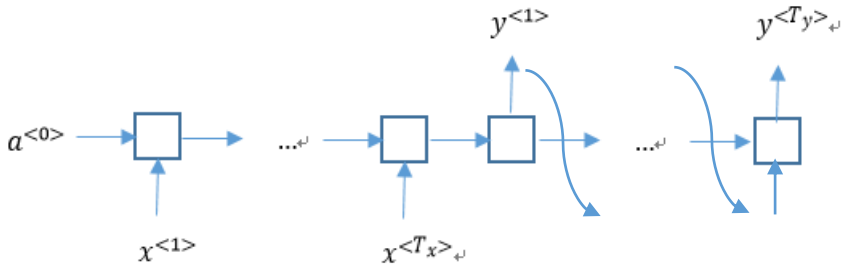
Sentiment classification
convert each word in sentence to e_x

"many2one" model



Beam search

parameter: Band width (B)



for any $y^{<i>}$, pick the most probable #B candidates.

$p(\hat{y}^{<i+1>}, \hat{y}^{<i>}|x) = p(\hat{y}^{<i>}|x) \cdot p(\hat{y}^{<i+1>}|x, \hat{y}^{<i>})$

target:

$$y^* = argmax_y \prod_{t=1}^{T_y} p(\hat{y}^{<t>}|x, \hat{y}^{<1>}, \dots, \hat{y}^{<t-1>})$$

$=> y^* = argmax_y \left(\frac{1}{T_y^\alpha}\right) \sum_{t=1}^{T_y} p(\hat{y}^{<t>}|x, \hat{y}^{<1>}, \dots, \hat{y}^{<t-1>})$ $0 \leq \alpha \leq 1$. (Improvement: Length normalization)

error analysis:

① $p(y^*|x) > p(\hat{y}|x)$, but beam search choose $\hat{y}$. Beam search is at fault. use larger B.

② $p(y^*|x) < p(\hat{y}|x)$, RNN is at fault. Use deeper layers or different architecture.

Bleu score

Given different standard translations, how to evaluate the $\hat{y}$?

on n-gram version:

$$p_n = \frac{\sum_{n-gram \in \hat{y}} count_c(n-gram)}{\sum_{n-gram \in \hat{y}} count(n-gram)}$$

$count_c$: max(n-gram in y)

eg. (n=2)

Version 1	The cat is on the hat.	
Version 2	There is a cat on the hat.	
$\hat{y}$	The cat the hat.	
	count	$count_c$
The cat	1	1
Cat the	1	0
The hat	1	1

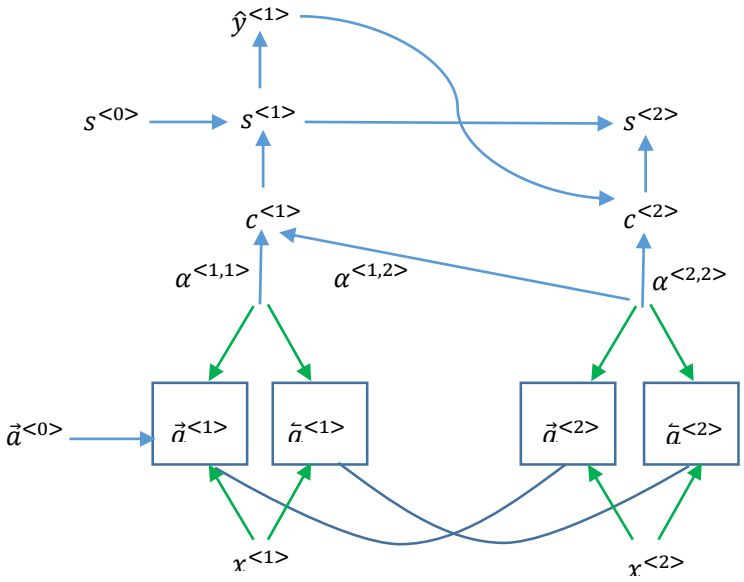
Combined Bleu score:

$$\text{BP} \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N} p_n\right)$$

$$\text{BP} = \begin{cases} 1 & len(\hat{y}) > len(y) \\ \exp\left(1 - \frac{len(\hat{y})}{len(y)}\right) & otherwise \end{cases}$$

Shorter $\hat{y}$ is more likely to have higher Bleu score, so give penalty.

Attention model



$$\begin{cases} \sum_{t'} \alpha^{<t,t'>} = 1 \\ c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \end{cases}$$

$\alpha^{<t,t'>}$ is the attention paid to $a^{<t'>}$ by $\hat{y}^{<t>}$.

Speech recognition

- CTC algorithm
- trigger word detection