CS 230: Deep Learning

Samuel Wong
Department of Statistics
Stanford University

Abstract

The course starts with an explanation of what a neural network is, deriving it from a simple logistic regression. It also covers gradient descent and some basic vectorization concepts. It then progresses to building deeper networks, hyperparameter tuning, and optimization algorithms. This is followed by a section on practical advice on how to implement a deep learning project. CNNs are then discussed, including popular open source CNN frameworks that can be used in transfer learning. Lastly, sequence models and embeddings are covered, including attention models.

Contents

1 Neural Networks and Deep Learning

1.1 Neural Network Basics

Binary Classification (0 or 1) Ex: cat or not cat

$$x = \begin{bmatrix} R1 \\ \vdots \\ R64 \\ G1 \\ \vdots \\ G64 \\ B1 \\ \vdots \\ B64 \end{bmatrix}$$

This is 64 x 64 x 3 = 12288 = n_x.
(x, y) = single datapoint; $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
m = number of training examples: $\{(x^1, y^1), (x^2, y^2), ..., (x^m, y^m)\}$

$$X = \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix}$$

$X \in \mathbb{R}^{n_x x m}$

$$Y = \begin{bmatrix} y^{(1)} & \dots & y^{(m)} \end{bmatrix}$$

$Y \in \mathbb{R}^{1xm}$
Logistic Regression:
Given x, want $\hat{y} = P(y = 1|x)$
Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$
Output: $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$
Loss function: $L(\hat{y}, y) = -(ylog\hat{y} + (1 - y)log(1 - \hat{y})$, where the loss is for 1 training example.
Cost function: $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[(ylog\hat{y} + (1 - y)log(1 - \hat{y})])$

1.2 Gradient Descent

Find w, b that minimize $J(w, b)$.
$w := w - \alpha\frac{\partial J(w,b)}{\partial w}$
$b := b - \alpha\frac{\partial J(w,b)}{\partial b}$
α = learning rate

Chain rule when backpropagating:
$\frac{\partial L(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$
$\frac{\partial L(a,y)}{\partial z} = \frac{\partial L}{\partial a} * \frac{\partial a}{\partial z} = [-\frac{y}{a} + \frac{1-y}{1-a}][a(1 - a)] = a - y$
$z = w_1x_1 + w_2x_2 + ... + w_nx_n = b$
$\frac{\partial L}{\partial w_1} = x_1\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$

With m examples:
$J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$
$\Rightarrow \frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial w_1}L(a^{(i)}, y^{(i)})$
average out all the derivatives to get the overall gradient to use for SGD

1.3 Python Usage

Vectorization:
Example: $w^T x$

Non-vectorized: for i in range(n_x): 2 += w[i] * x[i]
Vectorized: z = np.dot(w, x)

Some vectorized functions: np.exp(), np.log(), np.sum(), np.abs(), np.maximum(), v**2, 1/v (where v is a vector)

Broadcasting in Python:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$$

Do not use vectors in Python with shape (5,). Can use assert(a.shape == (5,1)) to check dimensions. Can reshape by a = a.reshape((5,1)) to reshape (5,) to (5,1).

1.4 Shallow/Deep Neural Neural Networks

Neural Network has an Input Layer, some Hidden Layers, and an Output Layer. When we talk about how many layers a NN has, we don't count the Input Layer.

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} — & w_1^T & — \\ — & w_2^T & — \\ — & w_3^T & — \\ — & w_4^T & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

With m training examples,

$$X = \begin{bmatrix} | & & | \\ x^1 & \dots & x^m \\ | & & | \end{bmatrix}$$

$Z^{[1]} = W^{[1]}X + b^{[1]}$
To calculate this, make sure to vectorize! No explicit for loops.

We can also use activation functions that are not the sigmoid function. Tanh is almost always better - only exception is for the output layer for a binary classification (since sigmoid gives a value between 0 and 1 and tanh gives one between -1 and 1). ReLU (max(0,z)) is also a great option.

Neural network needs a nonlinear activation function, otherwise the neural network just outputs a linear combination of the inputs.

Important to randomly initialize parameters, not just set it to 0. Otherwise all the nodes will do the same thing. For $W^{[1]}$, do a random value * 0.01 to keep it small to have faster SGD.

2 Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization

2.1 Train/Dev/Test

Traditional machine learning has train/dev split of 70/30 or maybe 60/20/20. However, if we have lots of data (say 1 Million), which may happen when we work with deep learning models, we can have splits of 98/1/1. Most importantly, we need to make sure that dev and test sets come from the same distribution, and if absolutely necessary, not having a test set may be ok.

2.2 Bias/Variance

High bias - underfitting; low training set error with high dev set error
High variance - overfitting; high training set error with high dev set error
We "compare" the error to optimal error (human error).

If high bias, we can proceed by:

1. using a bigger network
2. training longer (more iterations)
3. changing the NN architecture

If high variance, we can proceed by:

1. gathering more data
2. regularization
3. changing the NN architecture

If you have a high variance problem, you can use regularization.
$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})+$ regularization term
L2 regularization $\frac{\lambda}{2m}\|w\|_2^2$ (L2 regularization is also called weight decay).
L1 regularization $\frac{\lambda}{2m}\|w\|_1$ (L1 regularization will cause the weights to be sparse).
λ is the regularization hyperparameter.

Dropout regularization is another option $\rightarrow$ drop nodes at random during training (but not dropping out any nodes at test time). Most commonly used is the inverted dropout, where we set a "keep-prob" and divide "a" by the "keep-prob". We would typically have a lower "keep-prob" for earlier layers, and mostly keep for later layers.

Data augmentation can help a high variance problem (can use flipping, translation, distortion, color channel changing, etc.)

Can also use early stopping to solve high variance as well. Although the training error keep decreasing as we increase the number of iterations, the dev set error may start to increase at a certain point. We want to stop "early" before this happens.

2.3 Normalizing Training Sets

1. Subtract the mean: $x := x - \mu$
2. Normalize the variance: $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2$ element-wise, and then $x/ = \sigma^2$

We then use the same μ, σ^2 to normalize the test set.

2.4 Exploding/Vanishing Gradients

If $W^{[l]} > I$, then we can have exploding gradients.
If $W^{[l]} < I$, then we can have vanishing gradients.
We can help this problem with careful initialization of weights. Set $W^{[l]} =$ np.random.randn(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$). Then $\text{Var}(w_i) = \frac{2}{n}$ for ReLU. For tanh, we can use Xavier initialization: $\sqrt{\frac{1}{n^{[l-1]}}}$.

Note: we can do gradient checking to make sure we have made proper implementations: $\frac{f(\theta+\epsilon)-f(\theta-\epsilon)}{2\epsilon} \approx g(\theta)$

2.5 Optimization Algorithms

Mini-batch gradient descent:

- split training set into smaller training sets so you can take steps sooner rather than iterating over all training examples before taking one step
- split y's accordingly

1 epoch = 1 pass through the training set
1 epoch could be 5,000 gradient descent steps with mini-batch.
Stochastic gradient descent := minibatch of size 1

Exponentially weighted average: $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
Early stages of EWA can be biased strongly for the early terms. So we can correct it by using $\frac{V_t}{1-\beta^t}$

2.5.1 Gradient Descent with Momentum

$v_{dW} = \beta v_{dW} + (1-\beta)dW$
$v_{db} = \beta v_{db} + (1-\beta)db$
$W := W - \alpha v_{dW}$
$b := b - \alpha v_{db}$
Most common is to use $\beta = 0.9$

2.5.2 RMSProp (Root Mean Squared)

$s_{dW} = \beta s_{dW} + (1-\beta)dW^2$
$s_{db} = \beta s_{db} + (1-\beta)db^2$
$W := W - \alpha \frac{dW}{\sqrt{s_{dW}}}$
$b := b - \alpha \frac{db}{\sqrt{s_{db}}}$

2.5.3 Adam Optimization (combine Momentum and RMSProp)

$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$
$v_{db} = \beta_1 v_{db} + (1-\beta_1)db$
$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2$
$s_{db} = \beta_2 s_{db} + (1-\beta_2)db^2$

$v_{dW}^{corrected} = \frac{v_{dW}}{(1-\beta_1^t)}$
$v_{db}^{corrected} = \frac{v_{db}}{(1-\beta_1^t)}$
$s_{dW}^{corrected} = \frac{v_{dW}}{(1-\beta_2^t)}$
$s_{db}^{corrected} = \frac{v_{db}}{(1-\beta_2^t)}$

$W := W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon}$
$b := b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}} + \epsilon}$

Adam = adaptive moment estimation
$\alpha = \frac{1}{1 + decay_rate * epoch_num} * \alpha_0$, learning rate decay, gradually decreases over time.

2.5.4 Model Improvements

- Tuning hyperparamaters
  - Tune them in order of importance
  - Most important is the learning rate α
  - Next is number of hidden units, size of mini-batch
  - Try random values, don't grid search
  - Coarse to fine scheme
  - Use the appropriate scale for tuning (log scale or linear scale)

- train multiple models in parallel

- Batch normalization
  - Normalize z at every step
  - Makes contours more round so faster convergence
  - $\widetilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$, where β, γ learned by algorithm (parameters), and when using BatchNorm, no b's
  - At test time, estimate μ, σ^2 using exponentially weighted averages (across mini-batch) found during training to use at test time

- Use Softmax (for multi-class)
  - Last layer has softmax layer
  - Compute $t = e^{z^{[L]}}$
  - Compute $a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{n} t_i}$
  - $L(\hat{y}, y) = -\sum_{j=1}^{c} y_j log \hat{y}_j$

3 Structuring Machine Learning Projects

3.1 Introduction

Orthogonalization - each knob only changes one thing at a time

Issues with	Knobs
fit training set	bigger network, Adam, etc
fit dev set	regularization, bigger training set
fit test set	bigger dev set
real world	change dev set or cost function

Use a single number for evaluation metric. For example use F1 score instead of both precision and recall.

Optimizing vs Satisfying Metric: satisfying means that you're ok if it is good enough, which is secondary to the optimizing metric (only should have one of these).

We should get the dev and test sets from the same distribution, so randomly shuffle between them.

3.2 Comparing to Human Level Performance

- Model does not surpass Bayes optimal error
- If model is worse than humans, you can do the following
  - Get labeled data from humans
  - Gain insight from manual error analysis
  - Better analysis of bias/variance
- Human level error is a proxy for Bayer error

3.3 Error Analysis

- Examine a sample (say 100 misclassified examples and see where the model fails most and work on that). We get the most bang for buck doing this.
  - Could be misclassifying a particular type of image, in which case we can add more of these to the training set
  - Could be mislabeled dev data, so we may need to correctly label these items for the dev/test set
- Build first system quickly, then iterate
- Training/testing on different distributions
  - Sometimes this hpapens since there is not enough data (Ex: lots of web images, but not many mobile images to train mobile app)
  - Use majority labeled mobile data for dev/test, and the remaining for training set
  - Carve out a train-dev set out of training data to see if error is due to bias or variance

3.4 Transfer Learning

- Remove last layer and put in current task
- Initialize randomly the last layers weights and retrain (could just retrain the last 1 or 2 layers)

3.5 Other Learning Techniques

- Multi-task learning in order to augment data (train for multiple tasks at once)
- End to end deep learning
  - Works better than traditional pipeline only if you have large amounts of data

4 Convolutional Neural Networks

4.1 Introduction

For edge detection, we create a filter (such as a vertical edge detector) and then convolve it with the image. Example of a 3 x 3 vertical edge detector filter:

$$F = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

The model can learn the values in the filter through training, rather than a human pre-select them beforehand.

We can add padding so that the image does not keep shrinking as we add more layers, and also so that the pixels on the edges are used more.

- $n + 2p - f + 1$ will be the size of the next layer
- Valid convolution: no padding
- Same convolution: pad such that the output size = input size

We can add a stride s, which is how far we step each time, but that only takes a value if the entire filter lies within the image. The size of the next layer is then $\lfloor \frac{n+2p-f}{s} + 1 \rfloor$

We can do a convolution over a volume (say RGB image which has 3 layers). In that case, the depth is called the number of channels, and we can use multiple filters, and then stack the results to get the next layer.

4.2 CNN Layers

The different types of layers in a CNN are

- Convolutional (CONV)
- Pooling (POOL)
- Fully Connected (FC)

For a Maxpool (the most common type of POOL layer), the hyperparameters are the filter size (f) and the stride (s). There are no parameters to learn. If we have multiple layers, we do each channel separately and the size of the next layer is $\lfloor \frac{n_H - f}{s} + 1 \rfloor$

4.3 Famous CNN Architectures

- LeNet-5 (1998)
- AlexNet (2012)
- VGG16 (2015)
- ResNet (2015)
  - Helps you build much deeper networks
  - Helps with vanishing/exploding gradient problem by sending pushing earlier values into later layers: $a^{[l+2]} = g(z^{[l+2]} + z^{[l]})$
  - Fully Connected (FC)
- 1 x 1 Convolution (2013)
  - Also called Network on Network
  - Useful for shrinking channels from 1 layer to the next
  - Also useful for adding nonlinearity
- Inception (2014)
  - Stack different filters on top of each other (1 x 1, 3 x 3, 5 x 5, Maxpool etc)
  - To get dimensions to match, need to pad the maxpool
  - To reduce the compute cost, first use 1 x 1 convolution to reduce the number of channels, before running the 5 x 5 filters

4.4 Other Techniques

Transfer Learning

- Freeze all parameters and only retrain the last softmax layer
- Possibly the last couple of layers if we have more data

Data Augmentation

- Mirroring
- Random cropping
- Color shifting

4.5 Object Detection

4.5.1 Object Localization

- Not just detect a car, but put a bounding box around it
- Output not just a softmax for which type of object, but have 4 more numbers b_x, b_y, b_h, b_w for the bounding box

Assuming only 1 object at most:

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

where the b's indicate the location and the c's indicate which type of object it is.

$L(\hat{y}, y) = (\hat{y_1} - y_1)^2 + ... + (\hat{y_8} - y_8)^2$ if $y_1 = 1$, else if $y_1 = 0$, then $L(\hat{y}, y) = (\hat{y_1} - y_1)^2$

4.5.2 Object Detection

- First train ConvNet using closely cropped images, then do sliding window detection over the entire image
- Then repeat (region is large this time) etc.
- Sliding windows detection algorithm
- Disadvantage is computational cost (large strides may hurt performance)
- Can use convolutions instead of FC layers (5 x 5 x 16 $\rightarrow$ 400 is the same as 400 5 x 5 x 16 filters convolved)
- Can just run the full image through the CNN rather than the small ones 4 times
- Saves compute because you don't need to do duplicates

4.5.3 YOLO Algorithm (2015)

- Divide image into grid cells, and have all output from grid cells in the output. For example in an image that is split into 9 grid cells (3 x 3), assuming 8 features in each y (like the y shown above), you have 3 x 3 x 8 as the output y
- Each object can only be assigned into 1 grid cell
- b_h is the fraction of box height
- b_w is the fraction of box width
- b_x, b_y between 0 and 1

4.5.4 Object Detection Performance Analysis

- Intersection over Union (IoU)
  - Size of intersection of ground truth bounding box and model bounding box output / size of union of ground truth bounding box nad model bounding box output
  - By convention, "correct" if IoU ≥ 0.5
- Non-Max Suppression
  - Possible to have multiple grid cells claiming 1 for the same object
  - Looks at all "bounding boxes" with high IoU with the bounding box with the highest p_c and removes those, so remove redundancy
- What if grid cell needs to detect more than 1 object?
  - With two anchor boxes, each object in training is assigned to the grid cell and anchor box with highest IoU
  - Can allow for tall/skinny and wide/short boxes as well

5 Sequence Models

5.1 Introduction

$x^{(i)<t>}$ is the t^{th} element in the i^{th} training example
Cannot use standard network because

- Inputs/outputs different lengths in different examples
- Doesn't share features across different positions of text

5.2 Recurrent Neural Networks

$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$ for tanh/ReLU
$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y) for sigmoid/softmax$

Also, can rewrite as $a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ where $W_a = [W_{aa}\ W_{ax}]$
$L^{<t>}(\hat{y^{<t>}}, y^{<t>}) = -y^{<t>} log \hat{y}^{<t>} - (1 - y^{<t>}) log(1 - \hat{y}^{<t>})$
$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$

5.3 Language Modeling

Definition: given a sentence, what is the probability of that sentence $P(y^{<1>} ... y^{<T_y>})$
Training set: large corpus of English text
First step is to tokenize, and then can add ¡EOS¿ and ¡UNK¿ tokens and set $x^{<t>}$ as $y^{<t-1>}$
We can sample a sequence from a trained RNN to get a sense of what hte RNN has learned

- Sample randomly based on distribution learned by network
- Use $\hat{y}^{<1>}$ for $x^{<2>}$

Can also build a character level language model

Vanishing gradients with RNNs

- Very deep models can have problems updating earlier layers' weights, hard to capture long range dependencies - therefore use GRU or LSTM
- For exploding gradients, apply gradient clipping

5.4 Gated Recurrent Unit (GRU)

c = memory cell
$c^{<t>} = a^{<t>}$
$\widetilde{c^{<t>}} = tanh(w_c[c^{<t-1>}, x^{<t>}] + b_c)$
$\Gamma_u = \sigma(w_u[c^{<t-1>}, x^{<t>}] + b_u)$
$c^{<t>} = \Gamma_u * \widetilde{c^{<t>}} + (1 - \Gamma_u) * c^{<t-1>}$
Full GRU activation has:
$\widetilde{c^{<t>}} = tanh(w_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$
$\Gamma_r = \sigma(w_r[c^{<t-1>}, x^{<t>}] + b_r)$

5.5 Long Short Term Memory (LSTM)

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

We can also run a bidirectional RNN (BRNN) to gain info from later tokens.
Deep RNN - 3 layers is already a lot (rule of thumb).

5.6 Word Embeddings

- Can carry out transfer learning and word embeddings found online
- Can help with analogy relationships
- Can visualize using something like t-SNE to reduce from 300D to 2D
- Use cosine similarity: $sim(u,v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$

5.6.1 Word2Vec

- Skip-grams
  - Given a context word, try to predict a target word ± a certain distance
  - $o_c \rightarrow E \rightarrow e_c$ where $e_c = E_{o_c}$
  - $e_c uses a softmax$ where the softmax is $P(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$
  - to help improve speed for softmax, can use hierarchical softmax
- CBOW (continuous bag of words)
  - uses surrounding words to predict words
- Can use negative sampling
  - Pick context and target word not in actual and predict the target label (1 or 0)
  - $P(y=1|c,t) = \sigma(\theta_t^T e_c)$
  - Tuning an expensive 10000 way softmax into 10000 binary classification problems (where you're only updating 5 or so per iteration)

5.6.2 GloVe (Global Vectors for word representations)

- x_{ij} = number of times i (t) appears in the context of j (c)
- Minimize $\sum_{i=1}^{10000} \sum_{j=1}^{10000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - log X_{ij})^2$

5.6.3 Applications

- Can average embeddings of all the words in a review and feed it into a softmax for the score (1-5 stars)
- Better way is to use RNN with those embeddings
- There can be bias in word embeddings based on training corpus

5.7 Sequence to Sequence Models

- Encoding-decoding network for machine translation and image captioning
- For machine translation, select the most likely translation (do not sample at random!)
  - Select the sentence that is most likely, not the individual words that are more likely (so do not use greedy search)
  - Can use beam search for this (it is very expensive to finishing calculating all possible sentences before looking at probabilities), so pick B = 3 (consider on the 3 most likely phrases at the time)
  - Refinements to beam search
    - Maximize logs, not actual probabilities for numerical stability for really small numbers
    - Beam search favors shorter sentences so use $\frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} log P(y^{<t>}|x, y^{<1>}, ...y^{<t-1>})$ where in practice, $\alpha = 0.7$
  - Use the Bleu score as the primary metric
- To solve the problem with machine translation with long sentences, use an attention model

5.7.1 Attention Models

$\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

$\alpha^{<t,t'>} = \frac{exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} exp(e^{<t,t'>})}$

- Can use attention model for speech recognition
  - Separate info into many 1 sec clips etc, and collapse repeated characters not separated by blanks
  - For trigger word detection, use the same methodology, but label 1's for a few clips even after the trigger word to reduce the imbalance of 0's and 1's (have a higher proportion of 1's)