Technische Universität Berlin | Fakultät IV | Neuronale Informationsverarbeitung

Notes on Deep Learning Specialization

Dr. Rong Guo

Berlin, 2021

Contents

1 Introduction

1.1 Supervised learning

	Input (x)	Output (y)	Application
Standard NN	Home features	Price	Real estate
	Ad, user infor	Click on ad? (0/1)	Online Advertising
CNN	Image	Object $(1, 2, \cdots, 1000)$	Photo tagging
RNN	Audio	Text transcript	Speech recognitoion
	English	Chinese	Machine translation
Custom/Hybrid	Image, Radar info	Position of other cars	Autonomous driving

- Structured data
  each of the features, such as size of house, the number of bedrooms, or the age of a user, have a very well defined meaning
- Unstructured data
  Audio, Image, Text, etc.

1.2 Scale drives deep learning progress

- Data
- Computatios
- Algorithms
  e.g., activation function, sigmoid $\rightarrow$ reLU makes the gradient descent works much faster.

Idea $\rightarrow$ Code $\rightarrow$ Experiment

1.3 Notation

$(x, y), x \in \mathbb{R}^n, y \in \{0, 1\}$

m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

$m_{\text{train}}, m_{\text{test}}$

$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots x^{(m)} \end{bmatrix} \in \mathbb{R}^{n \times m}$ (stack x in column)

$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}$ (stack y in column)

To take the data associated with different training examples and to stack them in different columns.

1.4 Logistic regression

Given x, want $\hat{y} = p(y = 1|x), x \in \mathbb{R}^n, 0 \leq \hat{y} \leq 1$

Parameters: $w \in \mathbb{R}^n, b \in R$

Output: $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$

- If z large, $\sigma(z) \approx 1$
- If z large negative, $\sigma(z) \approx 0$

When programming neural networks, w and b are usually kept as separate parameters.

1.4.1 Logistic regression cost function

$\hat{y^{(i)}} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}, z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$, want $\hat{y} \approx y^{(i)}$.

Loss (error) function (w.r.t a single training example):
$L(\hat{y}, y) = -\left(y \log \hat{y} + (1 - y) \log (1 - \hat{y})\right)$

Cost function (w.r.t the whole training examples):
$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})\right)$

to minimize a convex function

1.5 Gradient descent

It starts at an initial point and then takes a step in the steepest downhill direction (steepest descent).
repeat until convergence {

$$w := w - \alpha \frac{\partial}{\partial w} J(w, b) \tag{1}$$

$$b := b - \alpha \frac{\partial}{\partial b} J(w, b) \tag{2}$$

}

2 One hidden layer neural network

2.1 Notation

- $$
\begin{aligned}
&a_i^{[l]} : l - \text{layer}; i - \text{node in layer} \\
&z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]}) \\
&z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]}) \\
&z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]}) \\
&z_3^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]}) \\
&W^{[1]} = \begin{bmatrix} -w_1^{[1]T} - \\ -w_2^{[1]T} - \\ -w_3^{[1]T} - \\ -w_4^{[1]T} - \end{bmatrix} \in \mathbb{R}^{\#(\text{hidden units}) \times n_x}
\end{aligned}
$$

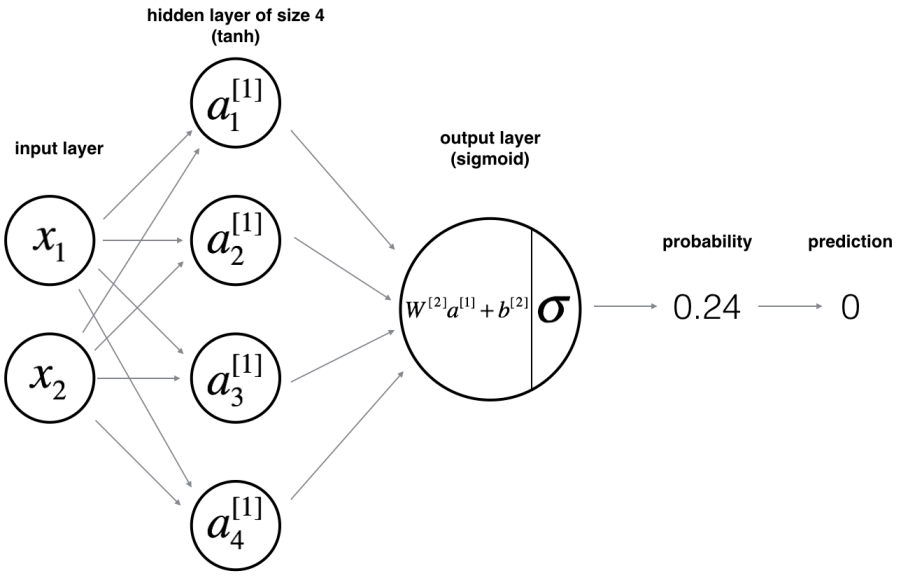

Figure 1: An example of an 1-hidden layer neural network

- For a single training example $x \to a^{[2]} = \hat{y}$

$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= \sigma(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]})
\end{aligned}
$$

- Vectorizing across multiple examples

$$a^{[l](i)} : l - \text{layer}; i - \text{example } i$$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_x \times m}$$

$$Z = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{\#(\text{hidden units}) \times m}$$

$$A = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{\#(\text{hidden units}) \times m}$$

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

2.2 Activation functions

1. $a = g(z) = \frac{1}{1+e^{-z}}$, sigmoid (except for the output layer of binary classification)
   $g'(z) = \dfrac{\mathrm{d}g(z)}{\mathrm{d}z} = g(z)(1 - g(z)) = a(1 - a)$
2. $a = g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
   $g'(z) = \dfrac{\mathrm{d}g(z)}{\mathrm{d}z} = 1 - (\tanh(z))^2 = 1 - a^2$
3. $g(z) = \max(0, z)$, ReLU, rectified linear unit

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases} \tag{3}$$

4. $a = \max(0.01z, z)$, leaky ReLU

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases} \tag{4}$$

2.3 Backpropagation

Forward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

Backpropagation

$$
\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

Initialize the weights randomly, e.g., $W^{[1]} = np.random.randn((2,2)) * 0.01, b^{[1]} = np.zeros((2,1))$, tanh *or sigmoid activation function usually prefers very small weights (large weights* $\rightarrow$ *large inputs* $\rightarrow$ *0 gradients). All zeros initialization leads to the symmetry breaking problem. In general, initializaing all the wights to zero results in the network failing to break symmetry. This means that every neuron in each layer will the the same thing, this way, the network is no more powerful than a linear classifier like logistic regression.*

3 Deep L-layer Neural Network

"shallow" vs. "deep"
logistic regression $\rightarrow$ 1 hidden layer $\rightarrow$ 2 hidden layers $\rightarrow$ L hidden layers

3.1 Notation

- l: # of layers
- $n^{[l]}$: # of units in layer l, $n^{[0]} = n_x$.
- $a^{[l]}$: activations in layer l, $a^{[l]} = g^{[l]}(z^{[l]})$.
  $w^{[l]}, b^{[l]}$: weights and bias for computing $z^{[l]}$.
- $x = a^{[0]}, \hat{y} = a^{[L]}$.

3.2 Forward propagation in a deep network

for a single training example:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \tag{5}$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \tag{6}$$

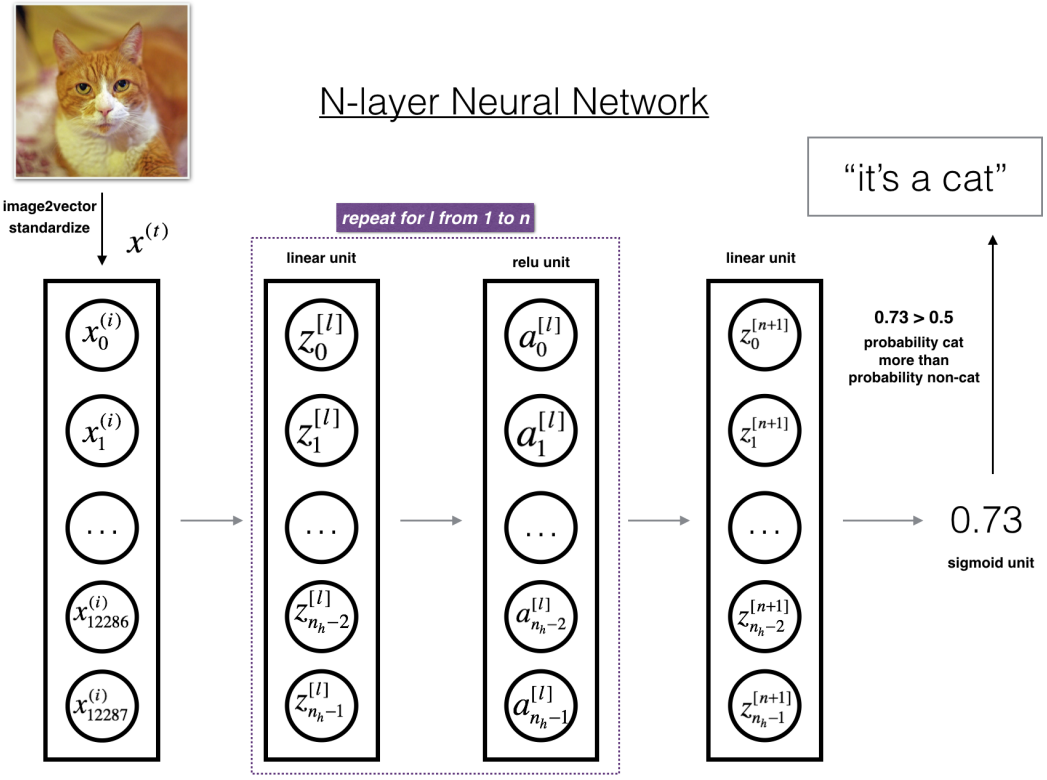

Figure 2: An example of an N-layer Neural Network: (LINEAR $\rightarrow$ RELU) $\times$ (L-1) $\rightarrow$ LINEAR$\rightarrow$ SIGMOID

dimensions of the matrices:

$$\begin{aligned} W^{[l]}, dW^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ b^{[l]}, db^{[l]} &: (n^{[l]}, 1) \\ z[l], a[l] &: (n^{[l]}, 1) \\ x, a^{[0]} &: (n^{[0]}, 1) \end{aligned}$$

vectorized for the whole training set (the training examples stacked in different columns):

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \tag{7}$$

$$A^{[l]} = g^{[l]}(Z^{[l]}) \tag{8}$$

dimensions of the matrices:

$$\begin{aligned} Z[l], A[l] &: (n^{[l]}, m) \\ dZ[l], dA[l] &: (n^{[l]}, m) \\ X, A^{[0]} &: (n^{[0]}, m) \\ W^{[l]}, dW^{[l]} &: (n^{[l]}, n^{[l-1]}) \\ b^{[l]}, db^{[l]} &: (n^{[l]}, 1), \text{broadcast to} (n^{[l]}, m) \end{aligned}$$

3.3 Backpropagation in a deep network

Forward and backward functions

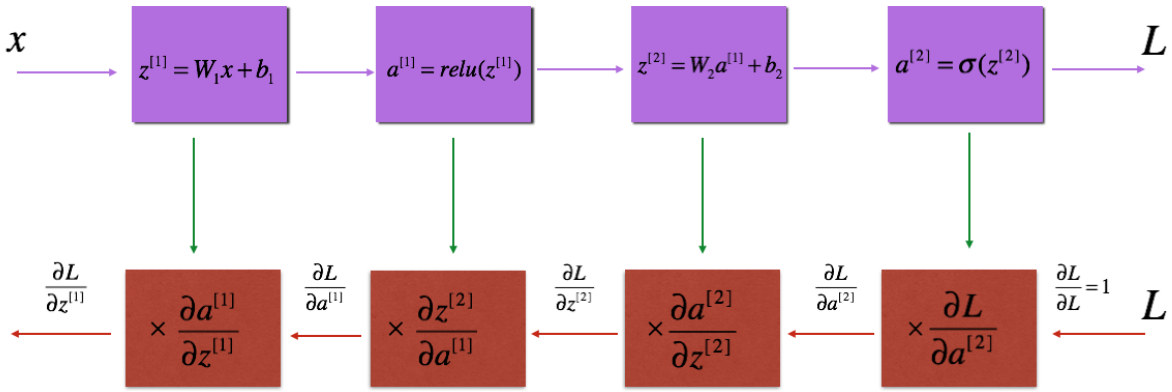

Figure 3: Basic blocks of forward and backward functions: $\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial W_1}$

Backpropagation

$$dZ^{[L]} = A^{[L]} - Y \tag{9}$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T} \tag{10}$$

$$db^{[L]} = \frac{1}{m} np.sum(dZ^{[L]}, axis = 1, keepdims = True) \tag{11}$$

$$dZ^{[L-1]} = W^{[L]T} dZ^{[L]} * g^{'[L-1]}(Z^{[L-1]}) \tag{12}$$

$$dZ^{[l]} = dA^{[l]} * g^{'[l]}(Z^{[l]}) \tag{13}$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{14}$$

$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True) \tag{15}$$

$$dA^{[l-1]} = W^{[l]T} dZ^{[l]} \tag{16}$$

Note that $*$ denotes element-wise multiplication[1].

3.4 Hyperparameters

- learning rate α
- # iterations
- # hidden layer L
- # hidden units $n^{[1]}, n^{[2]}, \cdots n^{[L]}$
- choice of activation function

1 For the cross-entropy cost $J = -\frac{1}{m}(y^{(i)} \log(a^{[L](i)}) + (1 - y^{(i)} \log(a^{[L](i)})))$, the gradient $dA^L = \frac{\mathrm{d}J}{\mathrm{d}A^L}$ is dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

- momentum, minibatch size, regulations, $\cdots$

Applied deep learning is a very empirical/iterative process (keep trying a lot of things and see what works): Idea → Code → Experiment; Cost J vs. # of iterations. Just have to try out many different values and go around the cycle.

4 Applied ML

- Train/dev/test sets
  Data → Training set;"Dev" (Hold-out cross validation, development set); Test
  Big data example: 98/1/1%; 99.5/0.4/0.1%
  Mismatched train/test distribution: make sure that the dev and test come from the same distribution. Not having a test set might be okay, i.e., only dev set. "train/dev" vs. "train/test".

- Bias / Variance
  - Train set error
  - Dev set error

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high variance, high bias	low variance, low bias

High bias? (training data performance) → *Bigger network*; Train longer; advanced optimization algorithms; NN architecture search.
High variance? (dev set performance) → *More data*; Regularization; NN architecture search
→ Low bias and low variance.

4.1 Regularization

4.1.1 Logistic regression

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 \tag{17}$$

$$\text{L2 regularization} : \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \tag{18}$$

$$\text{L1 regularization} : \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1 \tag{19}$$

λ = regularization parameters, which is usually set with the development set (cross validation). L1 regularization makes the model sparse, which is not very helpful for the purpose of compressing the model.

4.1.2 Neural networks

$$J(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2 \tag{20}$$

$$\text{Frobenius norm} : \|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (W_{i,j}^{[l]})^2 \tag{21}$$

"Weight decay":

$$\begin{aligned} dW^{[l]} &= \frac{1}{m} dZ^{[l]} A^{[l-1]T} + \frac{\lambda}{m} W^{[l]} \\ W^{[l]} &:= W^{[l]} - \alpha dW^{[l]} \\ &= W^{[l]} - \alpha(\frac{1}{m} dZ^{[l]} A^{[l-1]T} + \frac{\lambda}{m} W^{[l]}) \\ &= (1 - \frac{\alpha\lambda}{m}) W^{[l]} - \alpha(\frac{1}{m} dZ^{[l]} A^{[l-1]T}) \end{aligned}$$

The regularization term penalizes the weight being too large. With "weight decay", weights are pushed to smaller values.

4.1.3 Dropout regularization

Randomly knocking out units in the network: To go through each of the layers of the network and set some probability of eliminating a node in neural network. For each training example, train it using one of such smaller diminished networks. Dropout is only used during training (applied to both forward and backward propagation), NOT during test time.

Implementation "Inverted dropout".
Example: Illustrate with layer $l = 3$,

```
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3) # a3*= d3
a3 /= keep_prob # in order to not reduce the expected value of z4
```

Intuition: Can't rely on any one feature, so have to spread out weights → shrinking the weights, similar effect as to L2 regularization (an adaptive form of L2 regularization).

Downside: the cost function J is no longer well defined.

4.1.4 Other regularization methods

- Data augmentation, e.g., flipping horizontally, taking random crops/rotations/distortions/translations of the pictures to create additional fake training examples.
- Early stopping
  Plot the training error vs. # iterations and the dev set erorr vs. # iterations, stop half way to get mid-size $\|w\|_F^2$
  Downside: the tasks of optimizing cost function and reducing variance are no long orthogonalized (c.f., L2 regularization with trying out a lot of values of the hyperparameter λ).

4.2 The optimization problem

4.2.1 Normalizing training sets

(Rough intuition: the cost function will be faster and easier to optimize when the features are on similar scales).
Subtract mean and normalize variance:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$
$$x^{(i)} := \frac{x^{(i)} - \mu}{\sigma}$$

Use the same μ, σ^2 to normalize the test set.

4.2.2 Vanishing/Exploding gradients

Activation/gradients increase/decrease exponentially as a function of the number of layers.

$n \rightarrow$ careful choice of the random initialization of the neural network.

large $n \rightarrow$ smaller w
$W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$

ReLU: $Var(w) = \sqrt{\frac{2}{n^{[l-1]}}}$, He initialization
tanh : $Var(w) = \sqrt{\frac{1}{n^{[l-1]}}}$, Xavier initialization
$Var(w) = \sqrt{\frac{2}{n^{[l-1]} * n^{[l]}}}$

4.2.3 Gradient checking

Numerical approximation of gradients

$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \approx g(\theta)$$

Take all the $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.
Take all the $dW^{[1]}, db^{[1]}, \cdots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.
Is $d\theta$ the gradients of $J(\theta)$?

for each i:
$d\theta_{\text{approx}}[i] = \frac{J(\theta_1, \theta_2, \cdots, \theta_i + \epsilon, \cdots) - J(\theta_1, \theta_2, \cdots, \theta_i - \epsilon, \cdots)}{2\epsilon}$
check, e.g., $\frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \approx 10^{-7}$when $\epsilon \approx 10^{-7}$

- Don't use in training – only to debug.
- If algorithm fails the gradient checking, look at components to try to identify bug.
- Remember regularization.
  $J(\theta)$ includes the regularization term.

- Doesn't work with dropout.
- Run at random initialization; perhaps again after training.

5 Optimization algorithms

5.1 Mini-batch gradient descent

mini-batch: split the training data into smaller training sets, $X^{\{t\}}, Y^{\{t\}} \in \mathbb{R}^{n_x \times \frac{m}{n_{\text{mini-batch}}}}$
"1 epoch (a single pass through a mini-batch)": Forward propagation, compute the $J^{\{t\}}$ using $X^{\{t\}}, Y^{\{t\}}$, and update the parameters with backprob accordingly on each mini-batch.

Choosing the mini-batch size:

- If mini-batch size = m: Batch gradient descent → too long per iteration
- If mini-batch size = 1: Stochastic gradient descent → lose speedup from vectorization
  - If small training set: Use batch gradient descent ($m \leq 2000$)
  - Typical mini-batch size: 64, 128, 256, 512, 1024
  - Make sure that all the minibatch fit in CPU/GPU memory
  - Tune as another hyperparameter

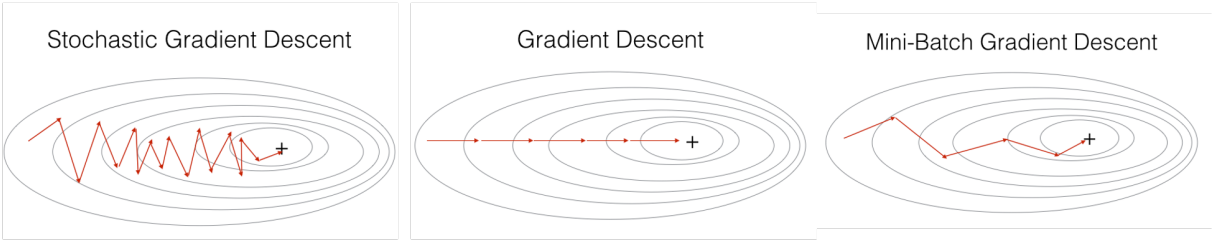

Figure 4: Comparison of different gradient descent methods

5.2 Exponentially weighted (moving) averages

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$
rule of thumb: $(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$
e.g., $0.9^{10} \approx 0.35 \approx \frac{1}{e}$
e.g., $0.98^{50} \approx \frac{1}{e}$

$v_\theta = 0$
Repeat{
  Get next θ_t
  $v_\theta := \beta v_\theta + (1-\beta)\theta_t$
}
Bias correction: $\dfrac{v_t}{1-\beta_t}$

damping out the oscillations in gradient descent and allowing the use of a larger learning rate to speed up the learning speed of the algorithm.

5.3 Gradient descent with momentum

Momentum
Momentum takes past gradients into account to smooth out the steps of gradient descent.
On iteration t:
Compute dw, db on current mini-batch.
$v_{dW} = \beta v_{dW} + (1-\beta)dW$
$v_{db} = \beta v_{db} + (1-\beta)db$
$W := W - \alpha v_{dW}, b := b - \alpha v_{db}$
Hyperparmeters: α, β, e.g., $\beta = 0.9$, average over about the 10 gradients.
The larger the mementum β is, the smoother the update, because it takes the past gradients into account more. But if β is too big, it could also smooth out the updates too much. Common values for β range from 0.8 to 0.999. Turing the optimal β might require several values to see what works best in terms of reducing the cost function J.

5.4 RMSprop

Root mean square prop
An exponentially weighted average of the squares of the past gradients, second moment estimates.
On iteration t:
Compute dw, db on current mini-batch.

$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2$ (element-wise squre)

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

$W := W - \alpha \frac{dW}{\sqrt{s_{dW}} + \epsilon}, b = b - \alpha \frac{db}{\sqrt{s_{db}} + \epsilon}$, e.g., $\epsilon = 10^{-8}$ numerical stability.

5.5 Adam optimization algorithm

Adam: Adaptive moment estimation
$s_{dW} = 0, s_{dW} = 0, v_{db} = 0, s_{db} = 0$
On iteration t:
Compute dw, db on current mini-batch.

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

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

$v_{dW}^{\text{corrected}} = v_{dW}/(1-\beta_1^t), v_{db}^{\text{corrected}} = v_{db}/(1-\beta_1^t)$

$s_{dW}^{\text{corrected}} = s_{dW}/(1-\beta_2^t), s_{db}^{\text{corrected}} = s_{db}/(1-\beta_2^t)$

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

Hyperparameters choice:
α: needs to be tuned
$\beta_1 : 0.9$
$\beta_2 : 0.999$
$\epsilon : 10^{-8}$

5.6 Learning rate decay

1 epoch: 1 pass through the whole training data

$$\alpha = \frac{1}{1 + \text{decayRate} \times \text{epochNumber}} \alpha_0$$

Other learning rate decay methods:

$\alpha = 0.95^{\text{epochNumber}} \alpha_0$
$\alpha = \frac{k}{\text{epochNumber}} \alpha_0$ or $\alpha = \frac{k}{t} \alpha_0$
$\alpha = \frac{1}{1 + \text{decayRate} \times \lfloor \frac{\text{epochNumber}}{\text{timeInterval}} \rfloor} \alpha_0$

manual decay, etc.

6 Improving deep neural networks

6.1 Hyperparameter tuning

- Hyperparameters: $\alpha, \beta, \beta_1, \beta_2, \epsilon$, # layers, #hidden units, learning rate dacay, mini-batch size
  Sampling randomly rather than using a grid, coarse to fine search process.

- Using an appropriate scale to pick hyperparameter:
  Using the log scale instead of a linear scale.

  $$r \in [\log_{10} 0.0001, \log_{10} 1] = [-4, 0]$$
  $$\alpha = 10^r$$

  Hyperparameters for exponentially weighted averages:
  $\beta = 0.9, \cdots, 0.999$
  sampling $1 - \beta$ instead of β using the log scale.

- Hyperparameters tuning in practice: babysitting one model vs. training many models in parllel.

6.2 Normalizing activations in a netowrk

Can we normalize $a^{[l]}$ so as to train $w^{[l+1]}, b^{[l+1]}$ faster?
Implementing batch norm:
Given some intermediate values in NN $z^{[l](i)}$,

$$\begin{aligned}
\mu &= \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)} \\
\sigma^2 &= \frac{1}{m}\sum_{i=1}^{m}(z^{[l](i)} - \mu)^2 \\
z^{[l](i)}_{\text{norm}} &= \frac{z^{[l](i)} - \mu}{\sigma + \epsilon} \\
\tilde{z}^{[l](i)} &= \gamma z^{[l](i)}_{\text{norm}} + \beta
\end{aligned}$$

γ, β are learnable parmeters of the model (avoiding the hidden units to always have mean 0 and variance 1), e.g., if $\gamma = \sigma + \epsilon, \beta = \mu$, then $\tilde{z}^{[l](i)} = z^{[l](i)}_{\text{norm}}$.

Working with mini-batches

$$X^{\{t\}} \xrightarrow{w^{[1]}} Z^{[1]} \xrightarrow[\text{BN}]{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \to a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{w^{[2]}} Z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \to a^{[2]} = g^{[2]}(\tilde{Z}^{[2]}) \xrightarrow{\cdots} \cdots$$

1. "covariates shift"
   It limits the amoun to which updating the parameters in the earlier layers can affect the distribution of values that the later layers have to learn on. It reduces the problem of the input values changing (stablize). It weakens the coupling between between the parameters of earlier and later layers.

2. "regularization effect"
   Each mini-batch is scaled by the mean/variance computed on just that mini-batch, which adds some noise to the values $z^{[l]}$ within that minimatch. So similar to dropout, it adds some noise to each hidden layer's activation. A bigger batch size will reduce the noise. But batch norm is not supposed to be used as regularization.

Working with testing set

μ, σ^2: estimate using exponentially weighted average (across mini-batch).

$$\begin{aligned}
&X^1 \to \mu^{\{1\}[l]} \\
&X^2 \to \mu^{\{2\}[l]} \\
&X^3 \to \mu^{\{3\}[l]} \\
&\quad\vdots \\
&\quad\to \\
&\mu(\text{exponentially weighted average})
\end{aligned}$$

If you are going to fine-tune a pretrained model, it is important that you block the weights of all your batchnormalization layers.

6.3 Multi-class classification: Softmax regression

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$t = e^{(z^{[l]})}(\text{element-wise exponentiation}) \tag{22}$$
$$a^{[l]} = g^{[l]}(z^{[l]}) = \frac{t_i}{\sum_{i=1}^{C} t_i} \tag{23}$$

When $C = 2$, the softmax reduces to the logistic regression.

6.3.1 Loss function

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j \log \hat{y}_j \tag{24}$$
$$J(w, b, \cdots) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{25}$$
$$(\text{maximum likelihood estimation}) \tag{26}$$

Gradient descent with softmax

$$\frac{\partial J}{\partial z^{[L]}} = \hat{y} - y$$

7 ML strategy

Collect more data; Collect more diverse training set; Train algorithm long with gradient descent; Try Adam instead of gradient descent; Try bigger network; Try smaller network; Try dropout; Add L_2 regularization; Network architecture (activation function, #hidden layers, $\cdots$); $\cdots$

7.1 Orthogonalization

What to tune in order to achieve one effect.
Chain of assumptions in ML:

Fit training set well on cost function: Bigger network, Adam, etc.

Fit dev set well on cost function: Regularization, etc.

Fit test set well on cost function: Bigger dev set, etc.

Performs well in real world: Change the dev set or the cost function, etc.

Orthogonalized controls make the process of tuning the network easier.

Single number evaluation metric
Satisficing and optimizing metric
N matices: 1 optimizing and N-1 satisficing.

7.2 Train/Dev/Test distributions

Taget: Dev set + metric.
Make sure that the dev set and the test set come from the same distribution.
→ Randomly shuffle the data into dev/test.
Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.

98% training, 1% dev, 1% training
Size of the test set: Set the test set to be big enough to give high confidence in the overall performance of the system.
When to change Dev/test sets and metrics? When the evaluation metric is no longer correctly rank ordering preferences between algorithms. If doing well on the metric + dev/test set does not correspond to doing well on the application, change the metric and/or dev/test set. e.g., $J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \rightarrow J = \frac{1}{\sum w^{(i)}}\sum_{i=1}^{m} w^{(i)} L(\hat{y}^{(i)}, y^{(i)})$

7.3 Comparing to human-level performance

Bayes optimal error (best possible error). Humans are quite good at a lot of tasks. So long as ML is worse than humans, you can:

Get labeled data from humans.

Gain insight from manual error analysis: Why did a person get this right?

Better analysis of bias/varaince.

Human-level error as a proxy for Bayes error. Understanding the estimate of Bayes error causes the decisions in choosing what to focus on.

- Human level error vs. Training error: "avoidable bias"
  → Train bigger model; Train longer/better optimization algorithms (e.g., RMSprop, Adam, etc.); NN architecture/hyperparameters search (e.g., RNN, CNN, etc.)
- Training error vs. Dev error: "variance"
  → More data; Regularization (e.g., L2, dropout, data augmentation, etc.); NN architecture/hyperparameters search

Problems where ML significantly surpasses human-level performance:

Online advertising

Product recommendations

Logistics (predicting transit time)

Loan approvals

Speech recognition

Some image recognition, medical data, etc.

The two fundatmental assumptions of supervised learning:

1. You can fit the training set pretty well
2. The training set performance generalizes pretty well to the dev/test set

8 Structuring ML projects

8.1 Error analysis

Look at dev examples to evaluate ideas
Check the "ceiling", upper bound on the performance, to evaluate whether or not a single idea is worth working on.
Evaluate multiple ideas in parallel.
→ a sense of the best options.

8.1.1 Cleaning up incorrectly labeled data

DL algorithms are quite robust to random errors in the training set (vs. "systematic errors"). Only if it makes a significant difference to evaluate the algorithms, it might worthwhile fixing the incorrect labels.
Example:
Overall dev set error: 10%
Errors due to incorrect labels: 0.6%
Errors due to other causes: 9.4%

> Apply same process to the dev and test sets to make sure they continue to come from the same distribution.
>
> Consider examining examples the algorithm got right as well as ones it got wrong.
>
> Train and dev/test data may now come from slightly different distributions. If the training set comes from a slightly different distribution, it might be just fine.

Build your first system quickly, then iterate. Set up dev/test and metric; Build initial system quickly; Use vias/variance analysis and error analysis to prioritize next steps.

8.2 Training and testing on different distributions

Example: Training set: 200,000; Test set: 10,000. → Training: 20,500; Dev 2,500; Test 2,500.

Example: Training set: 500,000; Test set: 20,000.
→ Training: 500,000; Dev 10,000; Test 10,000.
→ Training: 510,000; Dev 5,000; Test 5,000.

8.2.1 Bias and variance with mismatched data distributions

Training-dev set: Same distribution as training set, but not used for training.

Human error:	4%	
Training set error:	7%	→ avoidable bias
Training-dev set error:	10%	→ vairance
Dev set error:	12%	→ data mismatch
Test error:	12%	→ degree of overfitting

Addressing data mismatch

- Carry out manual error analysis to try to understand difference tween training and dev/test sets.
- Making the training data more similar; or collect more data similar to dev/test sets.

If artificial synthesis data are used, be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

8.3 Learning from multiple tasks

8.4 Transfer learning

Take knowledge the neural network has learned from one task and apply that knowledge to a separate task (sequential).
Having trained the neural network, to implement transfer learning: Swap in a new data set, replace the last output layer with a set of randomly initialized weights for the laster layer and have that output for the new task.

"pre-training" vs. "fine-tuning"
Depending on the size of the new dataset, re-train the last or the several later layer of the network.

When does transfer learning make sense?

- Task A and B have the same input x.
- A lot more data for task A than task B.
- Low level features from task A could be helpful for learning B.

8.5 Multi-task learning

start off simultaneously, trying to have one neural network do several things at the same time.
Example:
For $x^{(i)}, \hat{y}^{(i)} \in \mathbb{R}^{4\times 1}$,
$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y_j}^{(i)}, {y_j}^{(i)})$,
sum only over values of j with 0 or 1 label (each object doesn't have to be fully labeled).
Instead of training 4 separate neural networks, training one network to do 4 things. If some of

the earlier features in the network can be shared between these different types of objects, then training one network to do 4 things results in better performance.

Unlike softmax regression, which assigned a single label to single example. Each example has multiple labels. Building a single neural network that is looking at each example and solving multiple problems.

When does multi-task learning makes sense?

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually: Amount of data you have for each task is quite similar. e.g., 100 task and 1,000 examples per task.
- Can train a big enough neural network to do well on all the tasks.

8.6 End-to-end deep learning

Long complicated pipeline (multistep approach) $\rightarrow$ (input-end to output-end) train a huge neural network to just input the raw data and directly output the targets.

Speech recognition example
audio $\xrightarrow{\text{MFCC}}$ features $\xrightarrow{\text{ML}}$ Phonemes $\rightarrow$ Words $\rightarrow$ transcript
audio $\longrightarrow$ transcript

One of the challenges of end-to-end deep learning is that you might need a lot of data before it works well.

Pros and Cons of end-to-end deep learning

Pros:

- Let the data speak.
- Less hand-designing of components needed.

Cons:

- May need large amount of data.
- Excludes potentially useful hand-designed components (huge amount of data vs. hand-design)

Applying end-to-end deep learning
Key question: Do you have sufficient data to learn a function of the complexity needed to map x to y?

9 Convolutional neural networks

9.1 Computer vision problem

Image classification; Object detection; Neural style transfer.
Deep learning on large images: enough data to prevent a neural network from overfitting; computational/memory requirements infeasible.
In a computer vision application, each value in the matrix corresponds to a single pixel value. A filter is convolved with the image by multiplying its values element-wise with the original matrix, then summing them up and adding a bias.

"convolution" operation: filter (kernel) [2]
Example: Vertical edge detection:

$$\text{e.g., } \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Sobel filter:

$$\text{e.g., } \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Schorr filter:

$$\text{e.g., } \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

Example: Horizontal edge detection:

$$\text{e.g., } \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Let the numbers in the filter be learned (through backprop) as weights.

Padding

- It allows you to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "**same**" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image, e.g., zero padding. Without padding, very few values at the next layer would be affected by pixels at the edges of an image.
- "Valid" convolution:
  $(n \times n) * (f \times f) \to (n - f + 1) \times (n - f + 1)$
  Shrinking output, through away information from edge.
- "Same" convolution:
  Pad so that output size is the same as the input size

2 "Convolution" means something different in signal processing or math, cf. cross-correlation.

$\rightarrow (n + 2p - f + 1) \times (n + 2p - f + 1)$
e.g., pad the board with 1 or 2 pixels, $p = \frac{f-1}{2}$
f is usually odd.

Strided convolution
Stride = amount you move the window each time you slide padding p, stride s,
$(n \times n) * (f \times f) \rightarrow \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$

Convolutions over volume
Example: Convolutions on RGB images, e.g., hight, width, #chanels
$(6 \times 6 \times 3) * (3 \times 3 \times 3) \rightarrow (4 \times 4)$

Multiple filters
n'_c: # filters
$(n \times n \times n_c) * (f \times f \times n_c) \rightarrow (n - f + 1) \times (n - f + 1) \times n'_c$

9.2 One layer of a convolutional network

A convolution layer transforms an input volume into an output volume of different size.
Two filters

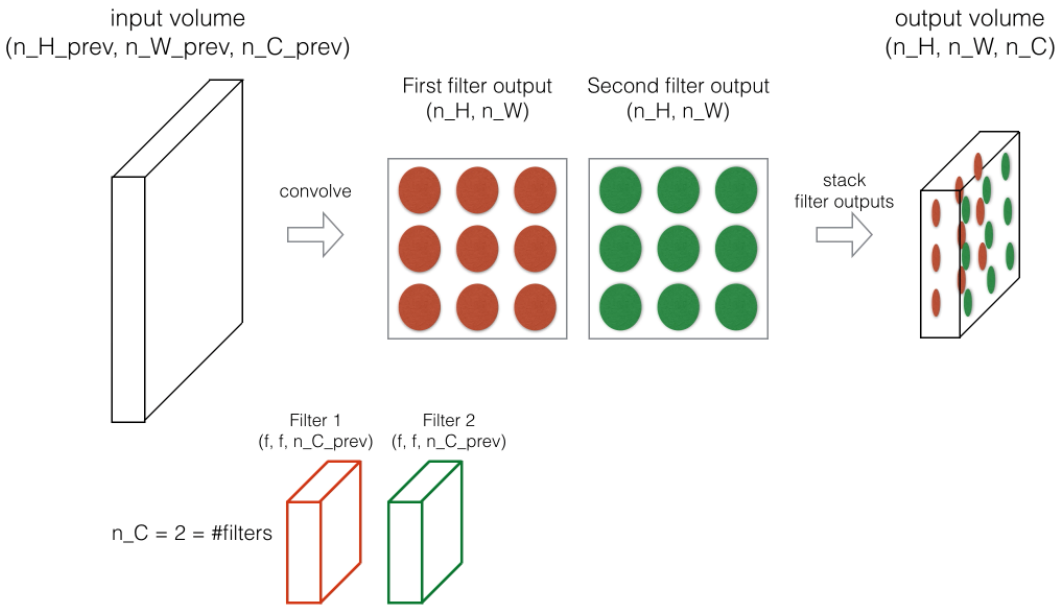

Figure 5: Illustration of convolutions

$(6 \times 6 \times 3) * (3 \times 3 \times 3) \rightarrow \text{ReLU}(4 \times 4 + b_1) \rightarrow (4 \times 4)$
$\underbrace{(6 \times 6 \times 3)}_{a^{[0]}} * \underbrace{(3 \times 3 \times 3)}_{w^{[1]}} \rightarrow \text{ReLU}(\underbrace{4 \times 4 + b_2}_{z^{[1]}}) \rightarrow (\underbrace{4 \times 4}_{a^{[1]}})$
$\rightarrow 4 \times 4 \times 2$

One property of such a convolution neural network: Less prone to overfitting. Once the feature detectors are learned, they could be applied to larger images and the number of

parameters keep fixed (relatively small).

If layer l is a convolution layer:
$f^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
$n_c^{[l]}$ = number of filters
Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$, $n_c^{[l]}$ is the number of filters in layer l
Bias: $n_c^{[l]} \rightarrow (1, 1, 1, n_c^{[l]})$

Input: $(n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]})$

Output: $(n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]})$

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$
$$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$$

9.3 Simple convolutional network example

Example ConvNet

x $(39 \times 39 \times 3)$:
$n_H^{[0]} = n_W^{[0]} = 39, n_c^{[0]} = 3$

$\longrightarrow$
$f^{[1]} = 3, s^{[1]} = 1, p^{[1]} = 0$, 10 filters
$n_H^{[1]} = n_W^{[1]} = 37, n_c^{[1]} = 10$

$a^{[1]}$ $(37 \times 37 \times 10)$:

$\longrightarrow$
$f^{[2]} = 5, s^{[2]} = 2, p^{[2]} = 0$, 20 filters
$n_H^{[2]} = n_W^{[2]} = 17, n_c^{[2]} = 20$

$a^{[2]}$ $(17 \times 17 \times 20)$:

$\longrightarrow$
$f^{[3]} = 5, s^{[3]} = 2, p^{[3]} = 0$, 40 filters
$n_H^{[3]} = n_W^{[3]} = 7, n_c^{[3]} = 40$

$a^{[3]}$ $(7 \times 7 \times 40)$
$\longrightarrow$
a vector of 1960 (flatten/unroll)

$\longrightarrow$
logistic regression/softmax

9.4 Types of layers in a convolutional network

- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

Pooling layer: Pooling layers gradually reduce the height and width of the input by sliding a 2D window over each specified region, then summarizing the features in that region.

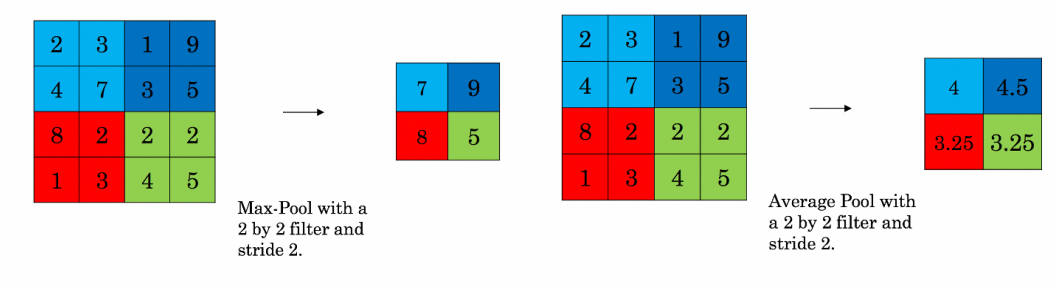

Figure 6: Max pooling (left) vs. average pooling (right)

- Max pooling:
  slides an (f, f) window over the input and stores the max value of the window in the output.
- Average pooling:
  slides an (f, f) window over the input and stores the average value of the window in the output.
- Hyperparameters (f: filter size; s: stride) but no parameters to learn.

e.g., $5 \times 5 \xrightarrow{f=3,s=1} 3 \times 3$
pooling is applied to each of the channel independently.
$n_H \times n_W \times n_c \rightarrow \left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times n_c$

9.5 CNN Example

Why convolutions:

- Parameters sharing:
  A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
- sparsity of connections:
  In each layer, each output value depends only on a small number of inputs.

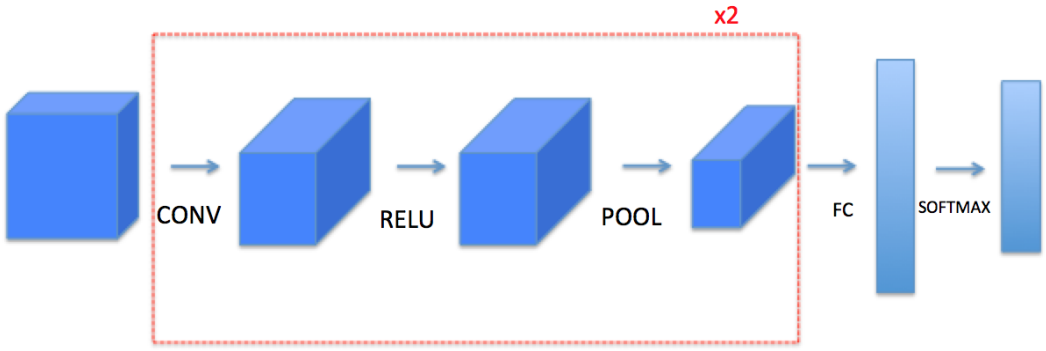

Figure 7: An example of CNN

Typically $n_H, n_W \downarrow; n_c \uparrow$

Training set $(x^{(1)}, x^{(1)}), \cdots, (x^{(m)}, x^{(m)})$
Cost $J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$
Use gradient descent to optimize parameters to reduce J.

CNN being good at translation invariance (e.g., a picture of a cat shifted a couple of pixels to the right, is still pretty clearly a cat). The convolutional structure helps the neural network encode the fact that an image shifted a few pixels should result in pretty similar features and should probably be assigned the same label.

10 Deep convolutional models

10.1 Classic networks

- LeNet
  LeCun et al., 1998. Gradient-based learning applied to document recognition
  $\sim$ 60K

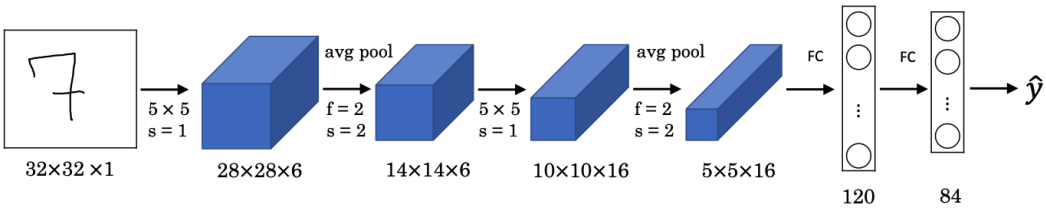

Figure 8: LeNet5

- AlexNet
  Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks
  $\sim$ 60M

- VGG-16
  CONV = 3×3 filter, s = 1, same MAX-POOL = 2×2 , s = 2

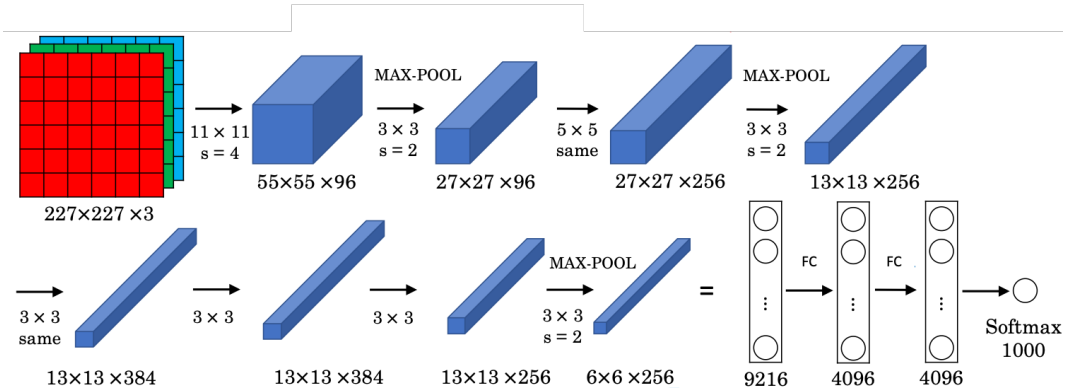

Figure 9: AlexNet

∼138M
Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition

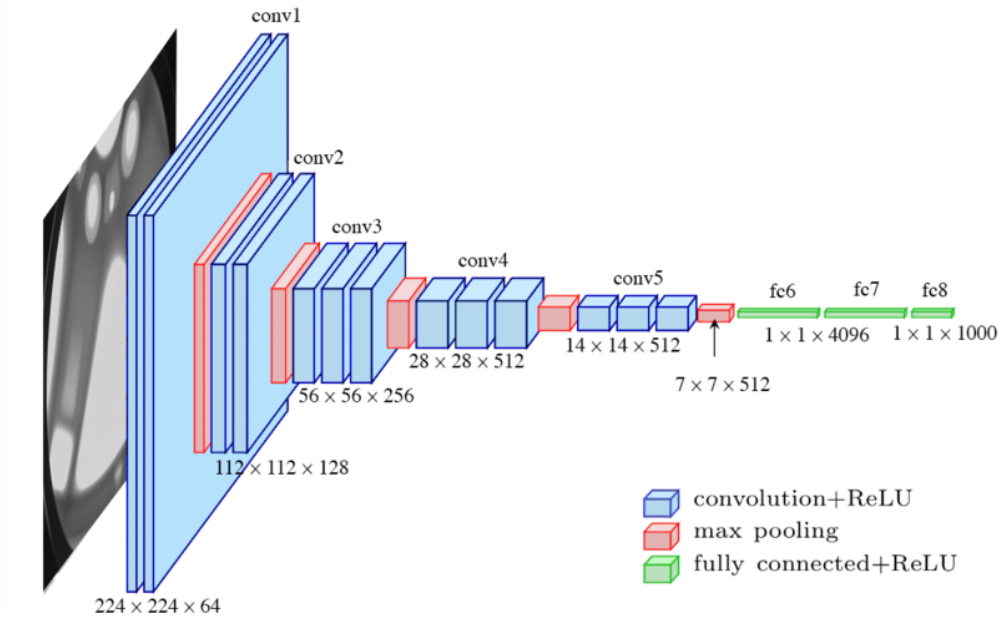

Figure 10: VGG16

- ResNets
  He et al., 2015. Deep residual networks for image recognition
  The motivation of Residual Networks is actually to help us train very deep networks. Indeed, very deep neural networks are hard to train and a deeper network does not always imply lower training error.
  - Residual block:
    "short cut"/skip connection vs. "main path"
    Two types of blocks:

  - identity block
  - convolutional block

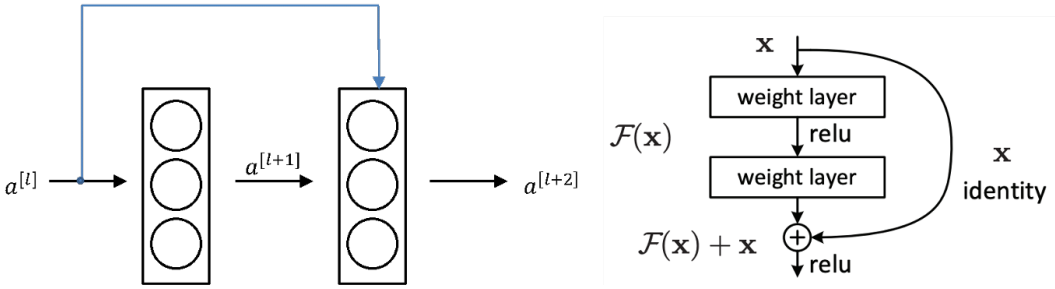

Figure 11: Residual block

  - "Plain network" → "Residual networks"
    which helps with the vanishing and exploding gradient problems and allows to train much deeper neural networks without really appreciable loss in performance.
  - Why do residual networks work?
    It's easy for those extra layers to learn the identity function, which kind of guaranteed not hurting the performance.

    In $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$, $z^{[l+2]}$ and $a^{[l]}$ suppose to have the same dimension, so very often the "same" convolutions are used. In case they have different dimensions, an extra weight matrix (W_s) is multiplied to $a^{[l]}$, or similarly a CONV layer in the shortcut path is used to resize the input to a different dimension, so that the dimensions match up in the final addition needed to add the shortcut value back to the main path. The CONV layer on the shortcut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.

- Inception Network (GoogLeNet)
  Lin et al., 2013. Network in network
  Szegedy et al. 2014. Going deeper with convolutions
  - Network in network (1×1 convolution)
    fully connected network applied to each of the positions across the channels → output # filters $n_c^{[l+1]}$
    to shrink the hight (n_H) and width (n_W) → pooling
    to shrink the number of channels (n_c) → 32 $1 \times 1 \times 192$ filters
    The 1×1 convolution can also keep n_c the same or increase it.
  - Inception module
    Previous activation → {CONV 1×1, CONV 1×1 + CONV 3×3, CONV 1×1 + CONV 5×5, MAXPOOL same, etc.} → Channel concat
    Instead of picking the filter sizes or pooling, do all of them and just concatenate the outputs (combination of the filters).
  - The problem of computational cost
    $28 \times 28 \times 192 \xrightarrow{\text{CONV5×5 same 32}} 28 \times 28 \times 32$

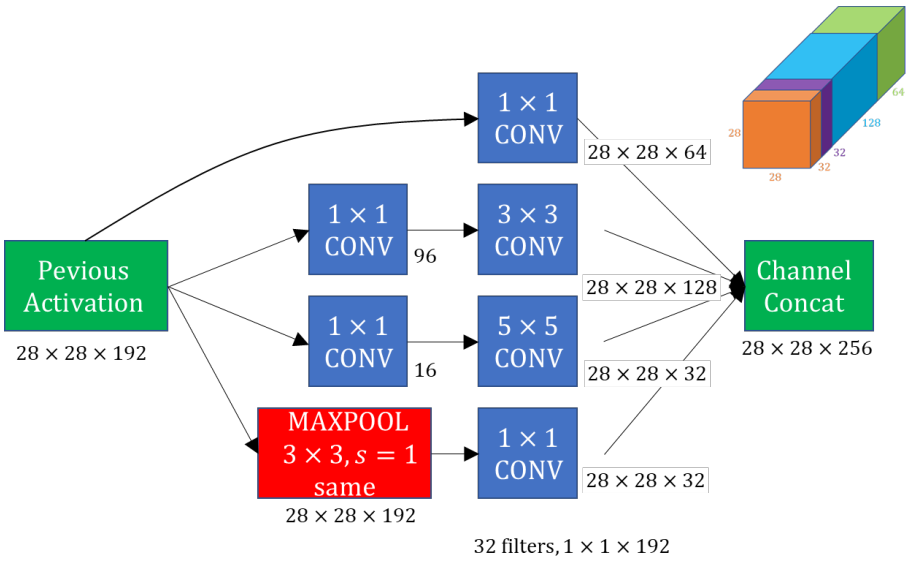

Figure 12: Inception module

Cost: 32 filters of $5 \times 5 \times 192 \rightarrow (28 \times 28 \times 32) \times (5 \times 5 \times 192) = 120\text{M}$
$28 \times 28 \times 192 \xrightarrow{\text{CONV1}\times1 \text{ same } 16} 28 \times 28 \times 16 \xrightarrow{\text{CONV5}\times5 \text{ same } 32} 28 \times 28 \times 32$, "bottleneck layer"
Cost: $(28 \times 28 \times 16) \times 192 + (28 \times 28 \times 32) \times (5 \times 5 \times 116) = 10\text{M}$

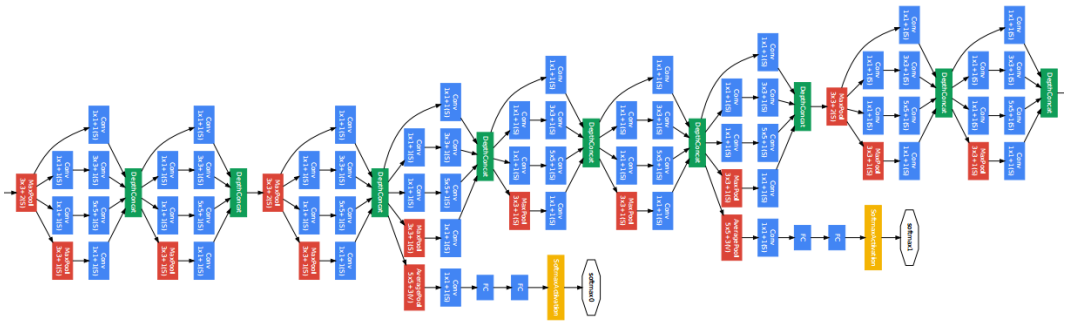

Figure 13: Inception network

10.2 MobileNet

- Low computational cost at deployment
- Useful for mobile and embedded vision applications
- Key idea: Normal vs. depthwise-separable convolutions
  - Normal convolution:
    $(n \times n \times n_c, 6 \times 6 \times 3) * (f \times f \times n_c \times n'_c, 3 \times 3 \times 3 \times 5) = (n_{out} \times n_{out} \times n'_c, 4 \times 4 \times 5)$
    Computational cost = #filter parameters × #filter positions × # filters
    $3 \times 3 \times 3 \times 4 \times 4 \times 5 = 2160$
  - depthwise-separable convolutions:
    Depthwise Convolution:

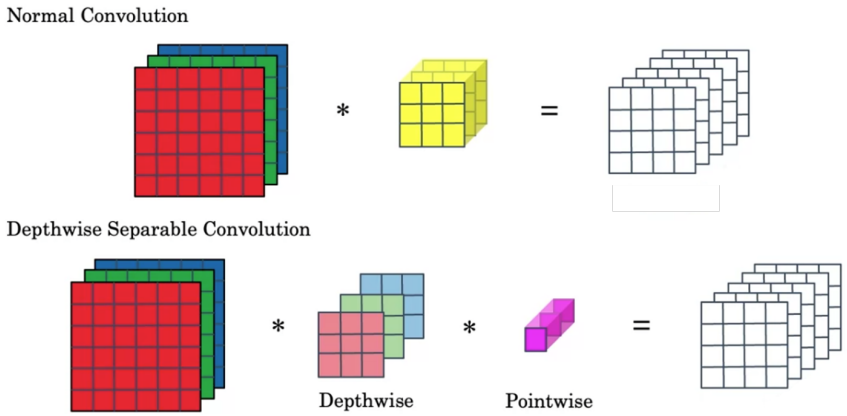

Figure 14: Normal vs. depthwise-separable convolution

$(n \times n \times n_c, 6 \times 6 \times 3) * (f \times f, 3 \times 3) = (n_{out} \times n_{out} \times n_c, 4 \times 4 \times 3)$
Computational cost = #filter parameters × #filter positions × # filters
$3 \times 3 \times 4 \times 4 \times 3 = 432$
Pointwise Convolution:
$(n_{out} \times n_{out} \times n_c, 4 \times 4 \times 3) * (f \times f \times n_c \times n'_c, 1 \times 1 \times 3 \times 5) = (n_{out} \times n_{out} \times n'_c, 4 \times 4 \times 5)$
Computational cost = #filter parameters × #filter positions × # filters
$1 \times 1 \times 3 \times 4 \times 4 \times 5 = 240$

- The ratio of the cost for the depthwise separable convolution compared to the normal convolution:
  $\frac{1}{n'_c} + \frac{1}{f^2}$

10.2.1 MobileNet architecture

MobileNet V1:
Input → 13 times depthwise convolutional operation → Pooling → FC → Softmax

MobileNet V2:
Input → 17 times Residual connections → Pooling → FC → Softmax
Residual connection (Bottleneck): Expansion → Depthwise → Projection (pointwise).
More computation inside the residual connection, then project back down to a smaller size outside the block, which allows the neural network to learn richer and more complex functions, while also keeping the amounts of memory that is the size of the activations passed from layer to layer, relatively small.

10.2.2 EfficientNet

r: resolution
d: depth
w: width
Given a particular computational budget, what's the good choice of r, d and w?
Baseline vs. Compound scaling, to scale up or down the neural networks based on the resources

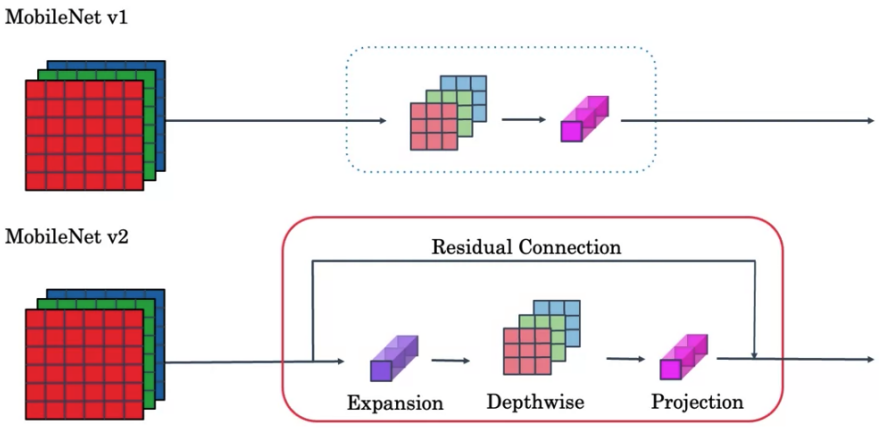

Figure 15: MobileNetV1 vs. MobileNetV2

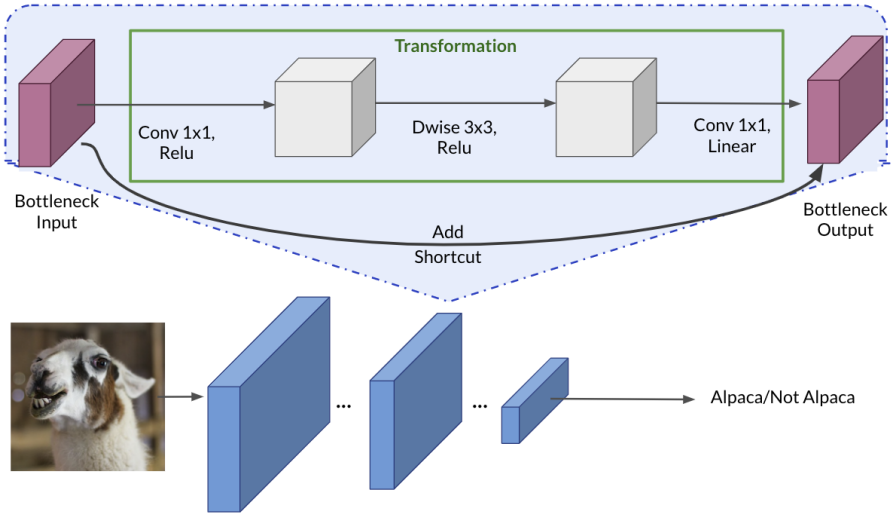

Figure 16: MobileNetV2 architecture

of a device.

10.3 Transfer learning

Rather than training the weights from scratch (random initialization), download weights that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in.
To adapt the classifier to new data: Delete the top layer, add a new classification layer, and train only on that layer. When freezing layers, avoid keeping track of statistics (like in the batch normalization layer). Fine-tune the final layers of your model to capture high-level details near the end of the network and potentially improve accuracy.

- Small dataset:
  Replace the softmax layer with your own application. Freeze the weights of all the previous layers, and re-train the weights at the softmax layer, e.g. trainable parameter, freeze, etc. in DL frameworks.

Pre-compute the activation of the layer before the softmax layer and save them to disk. Using this fixed function and then training a shallow softmax model from this feature vector to make a prediction.

- Large dataset:
  Freeze fewer (bigger data) earlier layers, and re-train weights of the later layers.
- A lot of data:
  Retrain the whole model.

10.4 Data Augmentation

Mirroring

Random Cropping

Rotation, Shearing, Local warping, etc.

Color shifting
PCA color augmentation

Implementing distortions during training

10.5 State of computer vision

Data vs. hand-engineering
Little data $\longrightarrow$ Lots of data
Objection detection Image recognition Speech recognition
More hand-engineering("hacks") $\longrightarrow$ Less hand-engineering (Simpler algorithms)

Two sources of knowledge

- Labeled data
- Hand engineered features / network architectures / other components

Tips for doing well on benchmarks/winning competitions

- Ensembling
  Train several networks (3 – 15) independently and average their outputs ($\hat{y}$).
- Multi-crop at test time
  Run classifier on multiple versions of test images and average results, e.g., 10–crop (1 central, 4 corners, 1 mirror central, 4 mirror corners).
- Use architectures of networks published in the literature
- Use open source implementations if possible
- Use pre-trained models and fine-tune on your dataset

11 Object Detection

What objects are in this image and where in the image are those objects located?

11.1 Object localization

What are localization and detection?
Image classification (1 object).
Classification with localization: figuring out where in the image is the object (1 object).
Detection: detect all the objects and localize them all (multiple objects maybe in different categories).

11.1.1 Classification with localization

Input → ConvNets → softmax (class labels) + bounding box
bounding box: b_x, b_y, b_w, b_h

Defining the target label y:

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}, \text{ e.g., } y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_w \\ b_h \\ 0 \\ 1 \\ 0 \end{bmatrix}, y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

The loss function (example with the SQE)

$$L(\hat{y}, y) = \begin{cases} \sum_{t=1}^{8} (\hat{y}_t - y_t)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases} \tag{27}$$

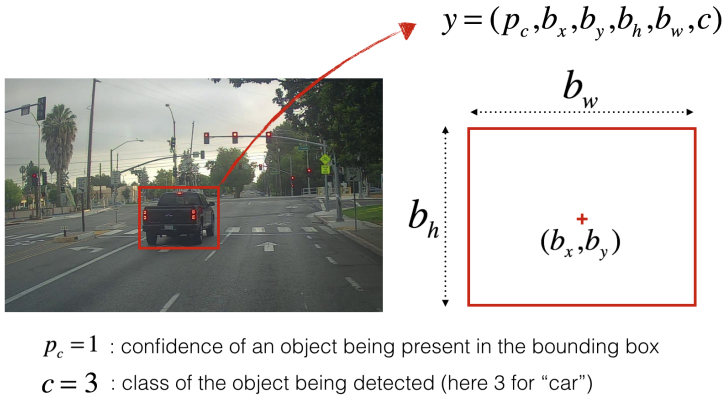

Figure 17: Definition of a box

11.1.2 Landmark detection

A neural network output X and Y coordinates of important points in an image, sometimes called landmarks, that you want the neural networks to recognize.
Example: face contour for AR. Input → ConvNets → 129 output units: face? $+ l_{1x}, l_{1y}, l_{2x}, l_{2y}, \cdots, l_{64x}, l_{64y}$.
Example: Pose/emotion detection.

11.2 Object detection algorithms

11.2.1 Sliding windows detection

Train a ConvNet on closely cropped examples of cars → Sliding windows detection (take a small rectangular region and input it to the ConvNet) → Go through the entire image → repeat with a larger window.

11.2.2 Convolutional implementation of sliding windows

Turning FC layer into convolutional layers.
Instead of doing it sequentially, implement convolutionally all at a time.

- $(14 \times 14 \times 3) \xrightarrow[5\times5]{} (10 \times 10 \times 16) \xrightarrow[2\times2]{\text{MAX POOL}} (5 \times 5 \times 16) \xrightarrow{\text{FC}} 400 \xrightarrow{\text{FC}} 400 \to y$ softmax(4)
- $(14\times14\times3) \xrightarrow[5\times5]{} (10\times10\times16) \xrightarrow[2\times2]{\text{MAX POOL}} (5\times5\times16) \xrightarrow[5\times5\times16\times400]{\text{FC}} 1\times1\times400 \xrightarrow[1\times1\times400\times400]{\text{FC}}$ $1 \times 1 \times 400 \xrightarrow[1\times1\times400\times4]{\text{FC}} y$ softmax$(1 \times 1 \times 4)$
- $(16\times16\times3) \xrightarrow[5\times5]{} (12\times12\times16) \xrightarrow[2\times2]{\text{MAX POOL}} (6\times6\times16) \xrightarrow[5\times5]{\text{FC}} 2\times2\times400 \xrightarrow[1\times1]{\text{FC}} 2\times2\times400 \xrightarrow[1\times1]{\text{FC}}$ y softmax$(2 \times 2 \times 4)$
- $(28 \times 28 \times 3) \xrightarrow[5\times5]{} (24 \times 24 \times 16) \xrightarrow[2\times2]{\text{MAX POOL}} (12 \times 12 \times 16) \xrightarrow[5\times5]{\text{FC}} 8 \times 8 \times 400 \xrightarrow[1\times1]{\text{FC}}$ $8 \times 8 \times 400 \xrightarrow[1\times1]{\text{FC}} y$ softmax$(8 \times 8 \times 4)$

Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks

11.2.3 Bounding box predictions: YOLO

Place down a grid on the input image, apply the image classification and localization algorithm to each of the grids.

Redmon et al., 2015, You Only Look Once: Unified real-time object detection
Example:

- Input image $(608, 608, 3)$
- The input image goes through a CNN, resulting in a $(19, 19, 5, 85)$ dimensional output.
- After flattening the last two dimensions, the output is a volume of shape (19, 19, 425):

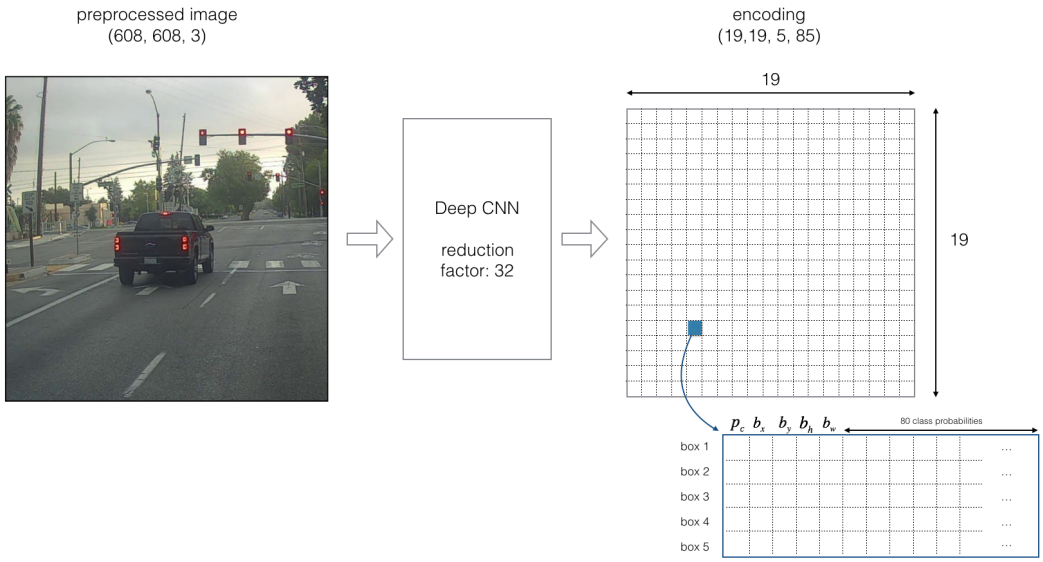

Figure 18: Encoding architecture for YOLO

  - Each cell in a 19×19 grid over the input image gives 425 numbers.
  - $425 = 5 \times 85$ because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes.
  - $85 = 5 + 80$ where 5 is because $(p_c, b_x, b_y, b_w, b_h)$ has 5 numbers, and 80 is the number of classes we'd like to detect

- select only few boxes based on:
  - Score-thresholding: throw away boxes that have detected a class with a score less than the threshold
  - Non-max suppression: Compute the Intersection over Union and avoid selecting overlapping boxes

Intersection over union (IoU)

IoU is a measure of the overlap between two bounding boxes.

$$\text{IoU} = \frac{\text{size of intersection}}{\text{size of union}}$$

"Correct" if IoU ≥ 0.5

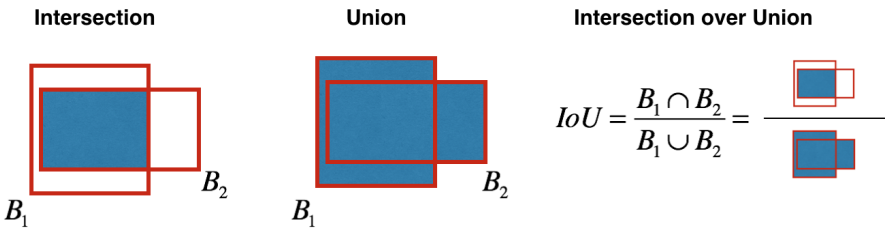

Figure 19: Definition of"Intersection over Union"

Non-max suppression example
Output the maximal probabilities but suppress the close-by ones that are non-maximal.

For multiple detections of each object, it looks at the probabilities associated with each of these detections (e.g., p_c).

1. Each output prediction is e.g., $y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_w \\ b_h \end{bmatrix}$

   Discard all boxes with $p_c \leq 0.6$

2. While there are any remaining boxes:
   - Pick the box with the largest p_c
     Output that as a prediction.
   - Discard any remaining box with IoU ≥ 0.5

Anchor boxes
Overlapping objects: Pre-define different anchor boxes.
With two anchor boxes:
Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU. (grid cell, anchor box)

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

Anchor box selection: To use K-means algorithm to group together types of objects' shapes.

11.2.4 Region proposal: R-CNN

Girshik et. al, 2013, Rich feature hierarchies for accurate object detection and semantic segmentation
Girshik, 2015. Fast R-CNN
Ren et. al, 2016. Faster R-CNN
Run segmentation algorithms to figure out what could be objects.

R-CNN
Propose regions. Classify proposed regions one at a time. Output label + bounding box.

Fast R-CNN
Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.

Faster R-CNN
Use convolutional network to propose regions.

11.3 Semantic segmentation with U-Net

Instead of just giving a single class label (and coordinates needed to specifying bounding box), to label every single pixel individually.
Per-pixel class labels.
Deep learning for semantic segmentation: Blow up small activation to bigger activation.

- Semantic image segmentation predicts a label for every single pixel in an image.
- U-Net uses an equal number of convolutional blocks and transposed convolutions for downsampling and upsampling.
- Skip connections are used to prevent border pixel information loss and overfitting in U-Net.

11.3.1 Transpose convolutions

place the filter on the output.

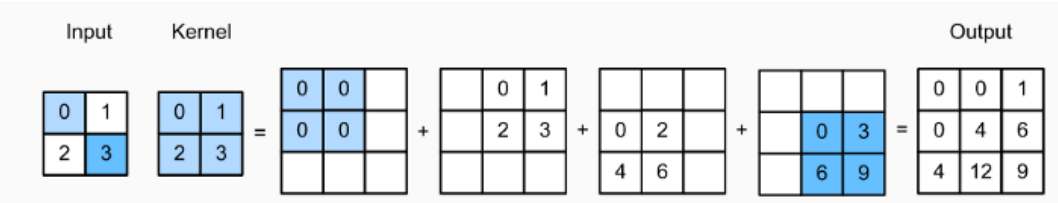

Figure 20: An example of transpose convolution computation

11.3.2 U-Net architecture intuition

U-net uses a matching number of convolutions for downsampling the input image to a feature map, and transposed convolutions for upsampling those maps back up to the original input image size.
It also adds *skip connections*, to retain information that would otherwise become lost during encoding. Skip connections send information to every upsampling layer in the decoder from the corresponding downsampling layer in the encoder, capturing finer information while also keeping computation low.
Ronneberger et al., 2015, U-Net: Convolutional Networks for Biomedical Image Segmentation

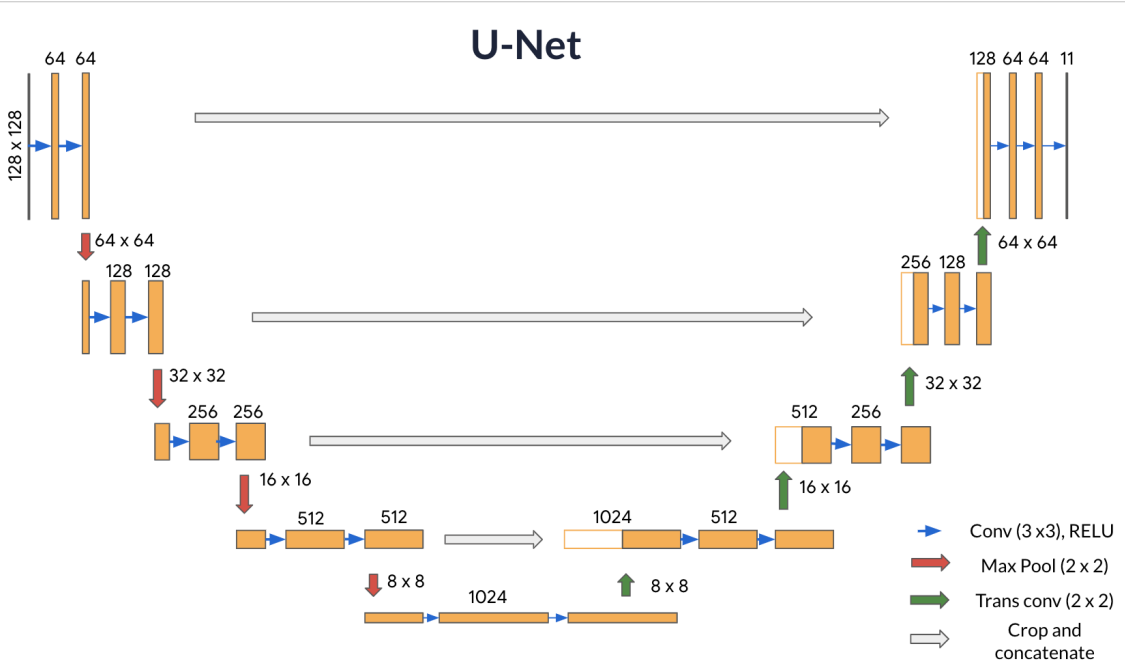

Figure 21: U-Net Architecture

12 Face recognition & Neural style transfer

12.1 Face recognition

Face verification vs. face recognition

- Verification
  - Input image, name/ID
  - Output whether the input image is that of the claimed person
- Recoginition
  - Has a database of K persons
  - Get an input image
  - Output ID if the image is any of the K persons (or "not recognized")

12.2 One shot learning

Learning from one example to recognize the person again.
Learning a "similarity" function
d(img1, img2) = degree of difference between images
Verification:
If d(img1, img2) $\leq \tau$: "same";
If d(img1, img2) $\geq \tau$: "different".

12.3 Siamese network

Taigman et. al., 2014. DeepFace closing the gap to human level performance
Two input images: $x^{(1)}, x^{(2)}$
"encoding" of the images: $f(x^{(1)}), f(x^{(2)})$
$d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2$

Parameters of NN define an encoding $f(x^{(1)})$
Learn parameters so that:
If $x^{(1)}, x^{(2)}$ are the same person, $\|f(x^{(1)}) - f(x^{(2)})\|_2^2$ is small
If $x^{(1)}, x^{(2)}$ are different persons, $\|f(x^{(1)}) - f(x^{(2)})\|_2^2$ is large

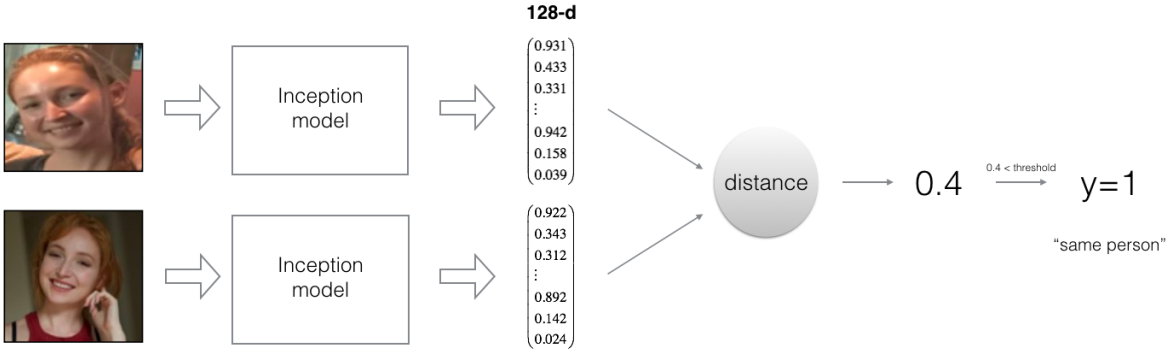

Figure 22: Computing the distance between two encodings and thrsholding

12.3.1 Triplet loss

Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering
Learning objective:
"push" the encodings of two images of the same person (Anchor and Positive) closer together, while "pulling" the encodings of two images of different persons (Anchor, Negative) further apart.

A is an "Anchor" image – a picture of a person.
P is a "Positive" image – a picture of the same person as the Anchor image.
N is a "Negative" image – a picture of a different person than the Anchor image.

$\|f(A) - f(P)\|^2 \leq \|f(A) - f(N)\|^2$
$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0$
α is the margin hyperparameter, which pushes the anchor-positive pair and the anchor-negative pair further away from each other. An image $A^{(i)}$ of an individual is closer to the Positive $P^{(i)}$ than to the Negative image $N^{(i)}$ by at least a margin α

Given triplets images, (A, P, N):
Loss function:

$$L(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0) \tag{28}$$

Cost function:
$J = \sum_{i=1}^{m} L(A^{(i)}, P^{(i)}, N^{(i)})$

Multiple pictures of the same person are needed, e.g., training set: 10k pictures of 1k persons.

Choosing the triplets (A, P, N)
During training, if (A, P, N) are chosen randomly, $d(A, P) + \alpha \leq d(A, N)$ is easily satisfied.
Choose triplets that are "hard" to train on.
It's only by choosing "hard" to triplets that the gradient descent procedure has to do some work to try to push these quantities further away from those quantities.

12.4 Face verification and binary classification

Taigman et. al., 2014. DeepFace closing the gap to human level performance
Input: two images.
Learning the similarity function: Take a pair of Siamese neural networks and have them both compute the embeddings, then have these be input to a logistic regression unit for a prediction.

$$\hat{y} = \sigma(\sum_{k=1}^{K} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b) \tag{29}$$

$$\hat{y} = \sigma(\sum_{k=1}^{K} w_k \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k} + b) \tag{30}$$

Pre-compute the encoding and then use that to make a prediction.

12.5 Neural style transfer

The optimization algorithm updates the pixel values rather than the neural network's parameters.

- An algorithm that given a content image and a style image can generate an artistic image.
- It uses representations (hidden layer activations) based on a pre-trained ConvNet.
- The content cost function is computed using one hidden layer's activations.
  The style cost function for one layer is computed using the Gram matrix of that layer's activations. The overall style cost function is obtained using several hidden layers.
- Optimizing the total cost function results in synthesizing new images.

Images: Content; Style; Generated image.

12.5.1 What are deep ConvNets learning?

Visualizing deep layers
Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks

The shallower layers of a ConvNet tend to detect lower-level features such as edges and

simple textures.
The deeper layers tend to detect higher-level features such as more complex textures and object classes.

12.5.2 Neural style transfer cost function

Gatys et al., 2015. A neural algorithm of artistic style

$$J(G) = \alpha J_{\text{Content}}(C, G) + \beta J_{\text{Style}}(C, G) \tag{31}$$

α and β are hyperparameters that control the relative weighting between content and style.

Find the generated image G

1. Initiate G randomly
   G: $100 \times 100 \times 3$
2. Use gradient descent to minimize $J(G)$
   $G := G - \frac{\partial J(G)}{\partial G}$
3. Content cost function
   - To choose hidden layer l to compute content cost (represent the content of an image).
     A layer in the middle of the network – neither too shallow nor too deep. This ensures that the network detects both higher level and lower level features.
   - Use pre-trained ConvNet. (e.g., VGG network)
   - Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
   - If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

   $$J^{[l]}_{\text{Content}}(C, G) = \frac{1}{2}\|a^{[l](C)} - a^{[l](G)}\|^2 \tag{32}$$

4. Style cost function
   - Say you are using layer l's activation to measure "style".
   - Define style as correlation between activations across channels.
     How correlated are the activations across different channels?
     How often the high-level features occur or don't occur together in different parts of an image (unnormalized cross covariance).

     Style matrix (Gram matrix [3]):
     Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k). $G^{[l]}$ is $n^{[l]}_c \times n^{[l]}_c$.

3 In linear algebra, the Gram matrix G of a set of vectors $(v_1, \cdots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j$. In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.

$G^{[l]}_{kk'}$ measures the correlation of the activations of channel k and k' at layer l, $k = 1, 2, \cdots, n^{[l]}_C$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'} \tag{33}$$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'} \tag{34}$$

$$J^{[l]}_{\text{Style}}(S, G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})^2 \tag{35}$$

$$J_{\text{Style}}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{\text{Style}}(S, G) \tag{36}$$

$$\sum_l \lambda^{[l]} = 1$$

"merge" style costs from several different layers. Each layer is given weights $\lambda^{[l]}$ that reflect how much each layer contributes to the style. This is in contrast to the content representation, where usually using just a single hidden layer is sufficient.

How do you choose the coefficients for each layer? The deeper layers capture higher-level concepts, and the features in the deeper layers are less localized in the image relative to each other. So if you want the generated image to softly follow the style image, try choosing larger weights for deeper layers and smaller weights for the first layers. In contrast, if you want the generated image to strongly follow the style image, try choosing smaller weights for deeper layers and larger weights for the first layers.

12.6 1D and 3D generalizations

2D convolution: $14 \times 14 \times 3 *$ filter $5 \times 5 \times 3 \times 16 \rightarrow 10 \times 10 \times 16$
1D convolution: $14 *$ filter $5 \rightarrow 10$ ($\times 16$ filters)
3D convolution: 3D volume * 3D filter, e.g., $14 \times 14 \times 14 \times n_c * 5 \times 5 \times 5 \times n_c \rightarrow 10 \times 10 \times 10 \times$ (16 filters)

13 Recurrent neural networks

Examples of sequence data:
Speech recognition
Music generation
Sentiment classification
DNA sequence analysis
Machine translation
Video activity recognition
Name entity recognition.

Notation:
NLP
Input: sequence of words;

Output: one output per input word, if the word is part of a person's name.
$x^{(i)\langle t\rangle}$: word at position t in the ith training example, at tth time step of a temporal sequence.
$y^{(i)\langle t\rangle}$: target for each word in the ith training example.
$T_x^{(i)}, T_y^{(i)}$: the length of the (input/output) sequence in the ith training example. Representing words: choosing a vocabulary/dictionary, then using one-hot representation (a vector of the length if the dictionary) to represent each of the words.

13.1 Recurrent neural network model

Why not a standard network?
Problems:

- Input, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text.

Basic RNN cell: Takes as input $x^{\langle t\rangle}$ (current input) and $a^{\langle t-1\rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t\rangle}$ which is given to the next RNN cell and also used to predict $\hat{y}^{\langle t\rangle}$.
A RNN is the repeated use of the single RNN cell. The time step dimension determines how many times to re-use the RNN cell.

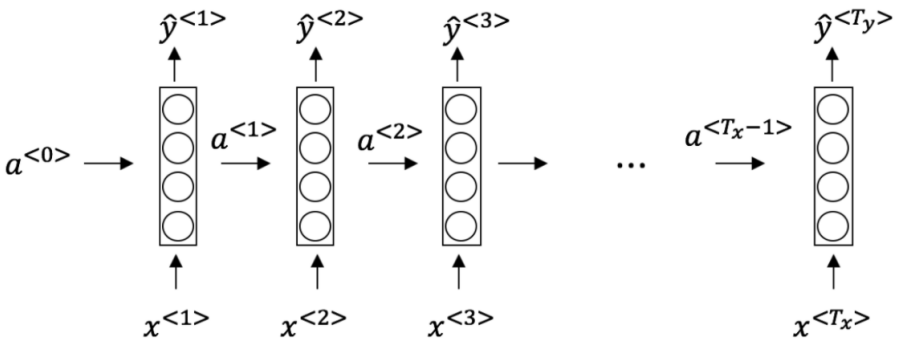

Figure 23: Architecture of a basic RNN ($T_x = T_y$)

One limitation of this particular unidirectional RNN is that the prediction at a certain time uses inputs/information from the inputs earlier in the sequence but not information later in the sequence → bi-directional RNN (BRNN).

Forward propagation Each cell takes two inputs at each time step: The hidden state

from the previous cell; The current time step's input data;
Each cell has two outputs at each time step: A hidden state; A prediction.

$$a^{\langle t\rangle} = g_1(w_{aa}a^{\langle t-1\rangle} + w_{ax}x^{\langle t\rangle} + b_a), \tanh/\text{ReLU} \tag{37}$$
$$y^{\langle t\rangle} = g_2(w_{ya}a^{\langle t\rangle} + b_y), \text{Sigmoid} \tag{38}$$
$$\rightarrow$$
$$a^{\langle t\rangle} = g_1(w_a[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_a) \tag{39}$$
$$y^{\langle t\rangle} = g_2(w_y a^{\langle t\rangle} + b_y) \tag{40}$$
$$w_a = [w_{aa}; w_{ax}]$$
$$[a^{\langle t-1\rangle}, x^{\langle t\rangle}] = \begin{bmatrix} a^{\langle t-1\rangle} \\ x^{\langle t\rangle} \end{bmatrix}$$

Backpropagation through time
Loss function (cross entropy loss):

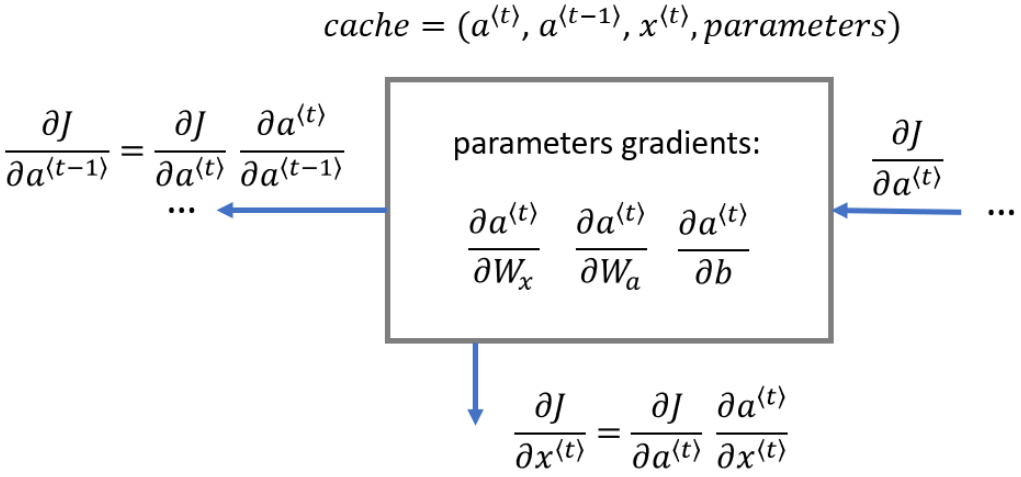

Figure 24: Backpropagation of an RNN cell

$$L^{\langle t\rangle}(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}) = -y^{\langle t\rangle}\log\hat{y}^{\langle t\rangle} - (1-y^{\langle t\rangle})\log(1-\hat{y}^{\langle t\rangle}) \tag{41}$$
$$L(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}) = \sum_{t=1}^{T_y} L^{\langle t\rangle}(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}) \tag{42}$$

In this case, the length of the input sequence was equal to the length of the output sequence (same number of time steps).

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a) \tag{43}$$
$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x) \tag{44}$$
$$dtanh = da_{next} * (1 - \tanh^2(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)) \tag{45}$$
$$dW_{ax} = dtanh \cdot x^{\langle t \rangle T} \tag{46}$$
$$dW_{aa} = dtanh \cdot a^{\langle t-1 \rangle T} \tag{47}$$
$$db_a = \sum_{batch} dtanh \tag{48}$$
$$dx^{\langle t \rangle} = {W_{ax}}^T \cdot dtanh \tag{49}$$
$$da_{prev} = {W_{aa}}^T \cdot dtanh \tag{50}$$
$$\tag{51}$$

13.1.1 Different types of RNNs

- $T_x = T_y$
  Many-to-many
  Machine translation
  $T_x \neq T_y$
- Sentiment classification
  x = text
  $y = 0/1$
  Many-to-one
- Music generation
  $x(x = \phi) \rightarrow y^{\langle 1 \rangle}y^{\langle 2 \rangle} \cdots y^{\langle T_y \rangle}$
  One-to-many
- *One-to-one*

13.1.2 Language model and sequence generation

Language model: p(sentence) = ? (output sentences that are likely)
Estimate $p(y^{\langle 1 \rangle}y^{\langle 2 \rangle} \cdots y^{\langle T_y \rangle})$

Language modelling with an RNN

- Training set: large corpus of english text.
  Tokenize; add <EOS> when a sentence ends; <UNK> for words not in the vocabulary.
- Input dummy vecotr of zeros $x^{\langle 1 \rangle} = \overrightarrow{0}, a^{\langle 0 \rangle} = \overrightarrow{0} \rightarrow \hat{y}^{\langle 1 \rangle}$ (length of the vocabulary way Softmax output: the probability of any word in the dictionary)

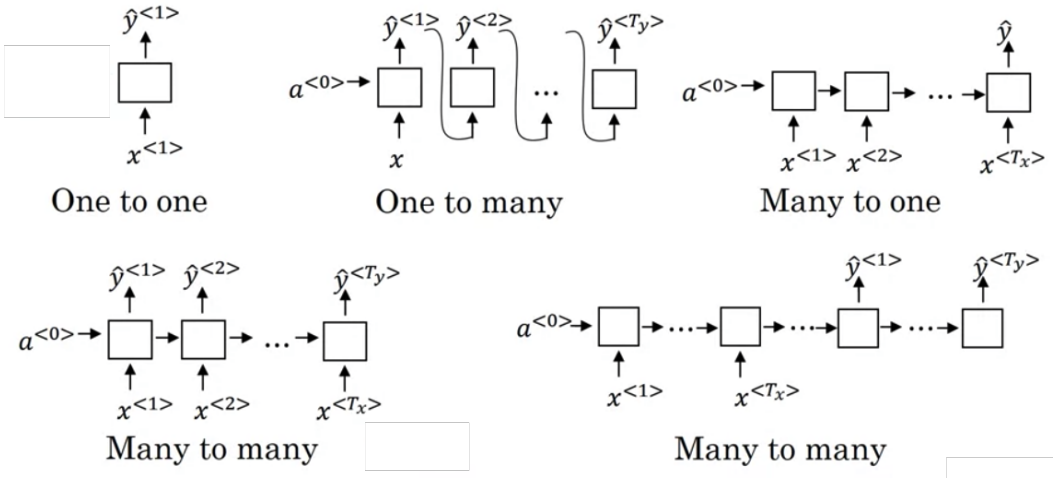

Figure 25: Architecture of different RNNs

$x^{\langle 2\rangle} = y^{\langle 1\rangle}, a^{\langle 1\rangle} \to p(\text{current word}|\text{proceeding words}) \cdots$. Run one step of forward propagation to get the first character and the probability distribution for the following character.

- Sampling a sequence from a trained RNN
  Randomly sample a word according to $\hat{y}^{\langle 1\rangle}$, pass it into the second and sample according to $\hat{y}^{\langle 2\rangle}$ given the procceding word (reject <UNK>), $\cdots$.

Character-level language model
$y^{\langle 1\rangle} y^{\langle 2\rangle} \cdots y^{\langle T_y\rangle}$ would be the individual character.
Pros: no unknown words tokens.
Cons: end up with much longer sentences.

13.1.3 Vanishing gradients with RNNs

Gradient vanishing (gradients decrease exponentially): The basic RNN has only "local" influence (information that is close to the prediction's time step t), not to be good at capturing long-range dependencies.
Gradient exploding problem (gradients increase exponentially): Gradient clipping.

13.2 Gated recurrent unit (GRU)

Cho et al., 2014. On the properties of neural machine translation: Encoder-decoder approaches
Chung et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

$c^{\langle t \rangle} = a^{\langle t \rangle} =$ memory cell

$$\tilde{c}^{\langle t \rangle} = \tanh(w_c[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c) \tag{52}$$
$$\Gamma_u = \sigma(w_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u), \approx \{0, 1\}, u - \text{update} \tag{53}$$
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle} \tag{54}$$

Full GRU

$$\tilde{c}^{\langle t \rangle} = \tanh(w_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c) \tag{55}$$
$$\Gamma_u = \sigma(w_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u) \tag{56}$$
$$\Gamma_r = \sigma(w_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r) \tag{57}$$
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle} \tag{58}$$
$$a^{\langle t \rangle} = c^{\langle t \rangle} \tag{59}$$

13.3 Long short term memory (LSTM)

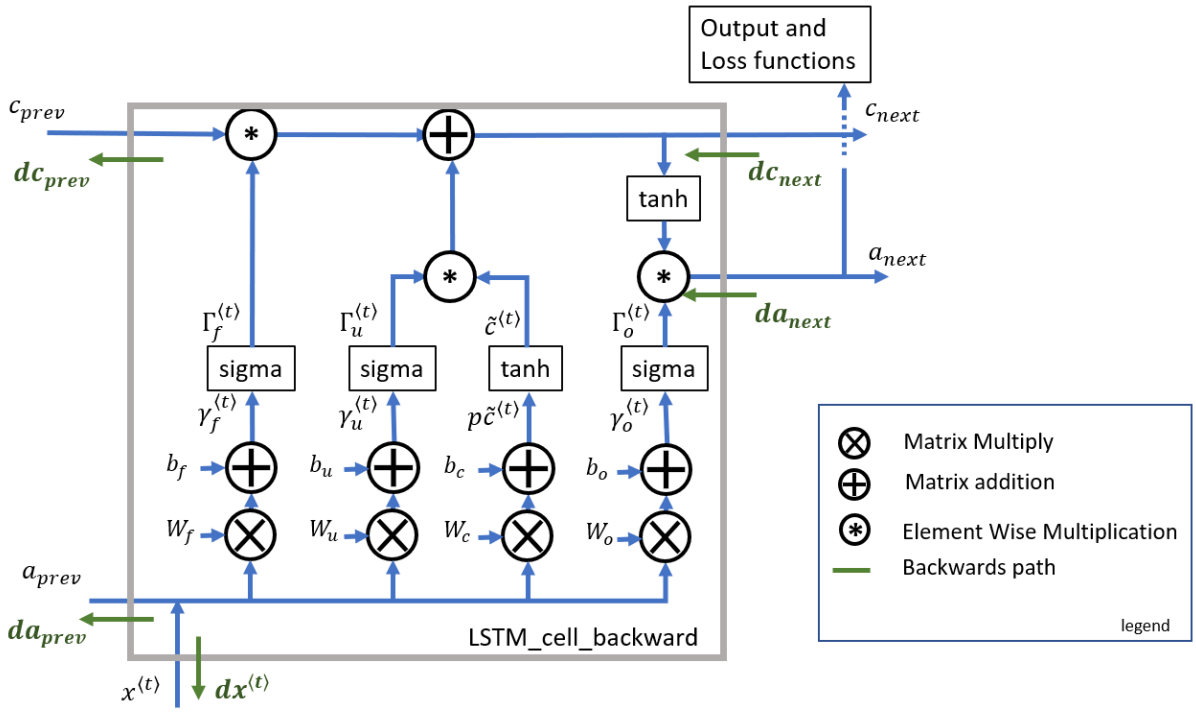

Figure 26: LSTM cell

Hochreiter & Schmidhuber 1997. Long short-term memory

$$\tilde{c}^{\langle t \rangle} = \tanh(w_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c) \tag{60}$$
$$\Gamma_u = \sigma(w_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u), \text{update gate}, \approx 0, 1 \tag{61}$$
$$\Gamma_f = \sigma(w_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f), \text{forget gate}, \approx 0, 1 \tag{62}$$
$$\Gamma_o = \sigma(w_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o), \text{output gate}, \approx 0, 1 \tag{63}$$
$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + \Gamma_f * c^{\langle t-1 \rangle} \tag{64}$$
$$a^{\langle t \rangle} = \Gamma_o * \tanh(c^{\langle t \rangle}) \tag{65}$$
$$\hat{y}^{\langle t \rangle} = \text{softmax}(w_y a^{\langle t \rangle} + b_y) \tag{66}$$

Multiplying the gates with the states element-wise (e.g., $Gamma_f * c^{\langle t-1 \rangle}$) is like applying a mask over the cell state.

An LSTM is similar to an RNN in that they both use hidden states to pass along information, but an LSTM also uses a cell state, which is like a long-term memory, to help deal with the issue of vanishing gradients. An LSTM cell consists of a cell state, or long-term memory, a hidden state, or short-term memory, along with 3 gates that constantly update the relevancy of its inputs.

LSTM as well as the GRU is very good at memorizing certain values, even for a long time for certain real values stored in the memory cells even for many, many times steps.
Peephole connection: add $c^{\langle t-1 \rangle}$ to each gate.

13.3.1 LSTM Backward Pass

Gate derivatives

$$d\gamma_o^{\langle t \rangle} = da_{next} * \tanh(c_{next}) * \Gamma_o^{\langle t \rangle} * \left(1 - \Gamma_o^{\langle t \rangle}\right) \tag{67}$$
$$dp\widetilde{c}^{\langle t \rangle} = \left(dc_{next} * \Gamma_u^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \Gamma_u^{\langle t \rangle} * da_{next}\right) * \left(1 - \left(\widetilde{c}^{\langle t \rangle}\right)^2\right) \tag{68}$$
$$d\gamma_u^{\langle t \rangle} = \left(dc_{next} * \widetilde{c}^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \widetilde{c}^{\langle t \rangle} * da_{next}\right) * \Gamma_u^{\langle t \rangle} * \left(1 - \Gamma_u^{\langle t \rangle}\right) \tag{69}$$
$$d\gamma_f^{\langle t \rangle} = \left(dc_{next} * c_{prev} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * c_{prev} * da_{next}\right) * \Gamma_f^{\langle t \rangle} * \left(1 - \Gamma_f^{\langle t \rangle}\right) \tag{70}$$

$$dW_f = d\gamma_f^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{71}$$
$$dW_u = d\gamma_u^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{72}$$
$$dW_c = dp\widetilde{c}^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{73}$$
$$dW_o = d\gamma_o^{\langle t \rangle} \begin{bmatrix} a_{prev} \\ x_t \end{bmatrix}^T \tag{74}$$

$$db_f = \sum_{batch} d\gamma_f^{\langle t \rangle} \tag{75}$$

$$db_u = \sum_{batch} d\gamma_u^{\langle t \rangle} \tag{76}$$

$$db_c = \sum_{batch} d\gamma_c^{\langle t \rangle} \tag{77}$$

$$db_o = \sum_{batch} d\gamma_o^{\langle t \rangle} \tag{78}$$

$$da_{prev} = W_f^T d\gamma_f^{\langle t \rangle} + W_u^T d\gamma_u^{\langle t \rangle} + W_c^T dp\tilde{c}^{\langle t \rangle} + W_o^T d\gamma_o^{\langle t \rangle} \tag{79}$$

$$dc_{prev} = dc_{next} * \Gamma_f^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh^2(c_{next})) * \Gamma_f^{\langle t \rangle} * da_{next} \tag{80}$$

$$dx^{\langle t \rangle} = W_f^T d\gamma_f^{\langle t \rangle} + W_u^T d\gamma_u^{\langle t \rangle} + W_c^T dp\tilde{c}^{\langle t \rangle} + W_o^T d\gamma_o^{\langle t \rangle} \tag{81}$$

13.4 Bidirectional RNN

At the point in time to take information from both earlier and later in the sequence. Acyclic

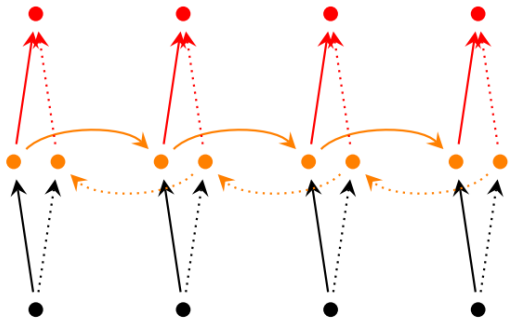

Figure 27: Bidirectional-RNN

graph

$$\hat{y}^{\langle t \rangle} = g(w_y[\overrightarrow{a}^{\langle t \rangle}, \overleftarrow{a}^{\langle t \rangle}] + b_y) \tag{82}$$

BRNN with LSTM

13.5 deep RNN

LSTM has a flag called "return sequences" to decide if you would like to return every hidden states or only the last one. You can use Dropout right after LSTM to regularize your network.

$$a^{[l]\langle t \rangle} = g(w_a^{[l]}[a^{[l]\langle t-1 \rangle}, a^{[l-1]\langle t \rangle}] + b_a^{[l]}) \tag{83}$$

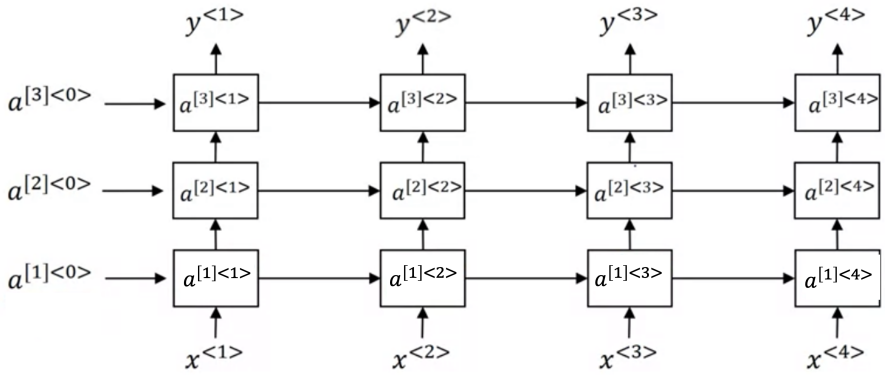

Figure 28: An deep RNN example

14 Natural Language Processing & Word Embeddings

14.1 Word representation

V = [a, aaron, ..., zulu, <UNK>]
1-hot representation

e.g., $O_{5391} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$

Such 1-hot representation doesn't allow the algorithm to easily generalize across words, because the product between any two different 1-hot vectors is 0.

Featurized representations: word embedding. Instead of a 1-hot presentation, to learn a featurized representation with each of the word, e.g., e_{5391}. This allows generalize much better across words, though the exact representations are harder to interpret.

embedding, visualizing word embeddings (i.e., featurized representations).
van der Maaten and Hinton., 2008. Visualizing data using t-SNE, non-linear mapping

Transfer learning and word embeddings

1. Learn word embeddings from large text corpus. (1-100B words), or down load pre-trained embedding online, e.g., pre-trained 50-dimensional GloVe embeddings.
2. Transfer embedding to new task with smaller training set, e.g., 100k words.
3. Optional: Continue to finetune the word embeddings with new data.

Relation to face encoding
learn a fixed embedding for each of the word in the fixed vocabulary.

Properties of word embeddings
Example: Analogies
Mikolov et. al., 2013, Linguistic regularities in continuous space word representations

$e_{\text{man}} - e_{\text{woman}} \approx e_{\text{king}} - e_{\text{w}}$
Find word w : $\underset{w}{\text{argmin}}$ similarity$(e_w, e_{\text{king}} - e_{\text{man}} + e_{\text{woman}})$

Cosine similarity
The cosine of the angle between two vectors is a measure of their similarity.

$$\text{CosineSimilarity(u, v)} = \frac{u \cdot v}{||u||_2||v||_2} = cos(\theta) \tag{84}$$

- $u \cdot v$ is the dot product (or inner product) of two vectors
- $||u||_2$ is the norm (or length) of the vector, $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$.
- θ is the angle between u and v.
- The cosine similarity depends on the angle between u and v.
  - If u and v are very similar, their cosine similarity will be close to 1.
  - If they are dissimilar, the cosine similarity will take a smaller value.

Cosine similarity is a good way to compare the similarity between pairs of word vectors. Note that L2 (Euclidean) distance also works. For NLP applications, using a pre-trained set of word vectors is often a great way to get started.

14.2 Learning word embeddings

14.2.1 Word2Vec

Embedding matrix E: $E \cdot O_{6257} = e_{6257}$
In practice, use specialized function to look up an embedding.
Bengio et. al., 2003, A neural probabilistic language model
Context/target pairs
Context:

- Natural language model: Last 4 words.
- Word embedding: 4 words on left & right; Last 1 word.

An embedding layer can be initialized with pretrained embedding matrix.

14.2.2 Skip-grams

Mikolov et. al., 2013. Efficient estimation of word representations in vector space.

Vocab size = 10,000

500-dimensional word embeddings
Mapping: Context c ("orange") → Target t ("juice")

$$O_c \to E \to e_c \to \text{softmax unit} \to \hat{y} \tag{85}$$

Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$
θ_t = parameter associated with output t
θ_t and e_c are both 500 dimensional vectors (initialized randomly at the beginning of training), trained with an optimization algorithm such as Adam or gradient descent. It is okay if the algorithm does poorly on this artificial prediction task; the more important by-product of this task is a useful set of word embeddings.
$L(\hat{y}, y) = -\sum_{j=1}^{10,000} y_i \log \hat{y}_i$
$y = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ is the 1-hot representation of the target.
Computation cost (the softmax part is very computationally heavy) → Hierarchical softmax

How to sample the context c?
Uniformly random → the, of, a, and, to, etc.
Balance out sampling with respect to $p(c)$.

14.2.3 Negative sampling

Mikolov et. al., 2013. Distributed representation of words and phrases and their compositionality
Defining a new learning problem

Context c	Word t	Target y
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

One positive example and k randomly selected negative examples. $p(y = 1|c, t) = \sigma(\theta_t^T e_c)$
10,000-way softmax → 10,000 binary logistic classification problems, on every iteration only train k+1 of them (k negative examples).

Selecting negative examples
Heuristic $p(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10,000} f(w_j)^{3/4}}$, which is proportional to the empirical frequency $f(w_i)$.

14.2.4 GloVe word vectors

Pennington et. al., 2014. GloVe: Global vectors for word representation
x_{ij} = # times $j(t)$ appears in context of $i(c)$.

$$\min \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(x_{ij})(\theta_i^T e_j + b_i + b'_j - \log x_{ij})^2 \tag{86}$$

weighting term:

$$f(x_{ij}) = \begin{cases} 0 \text{ if } x_{ij} = 0 \\ \text{heuristics for (in)frequent words} \end{cases}$$

$\theta_i^T e_j = (A\theta_i)^T (A^{-T}\theta_i)$. The individual components of the embeddings might not be interpretable. The axis used to represent the features might not be aligned with that night be easily humanly interpretable axis (potentially arbitrary linear transformation of the features).

14.3 Sentiment classification

Sentiment classification problem
x (a piece of texts) $\longrightarrow$ y (positive, negative, e.g., stars)

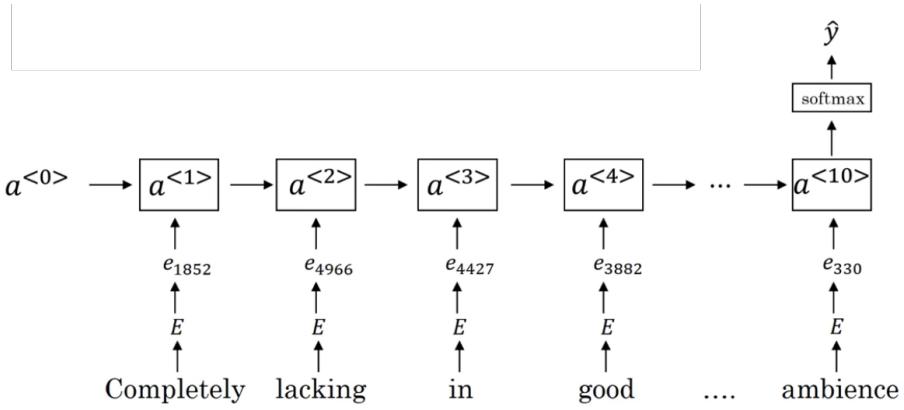

Figure 29: RNN for sentiment classification

14.4 Debiasing word embeddings

The problem of bias in word embeddings
Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embedding
Word embeddings can reflect gender, ethnicity, age, sexual, orientation, and other biases of the text used to train the model.
Addressing bias in word embeddings

1. Identify bias direction.
2. Neutralize: For every word that is not definitional, project to get rid fo bias.

$$e^{bias\_component} = \frac{e \cdot g}{||g||_2^2} * g$$

$$e^{debiased} = e - e^{bias\_component}$$

$e^{bias\_component}$ as the projection of e onto the direction g.

3. Equalize pairs.

15 Sequence to sequence models

- Machine translation
  Sutskever et al., 2014. Sequence to sequence learning with neural networks
  Cho et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation
- Image caption
  Mao et. al., 2014. Deep captioning with multimodal recurrent neural networks
  Vinyals et. al., 2014. Show and tell: Neural image caption generator
  Karpathy and Li, 2015. Deep visual-semantic alignments for generating image descriptions

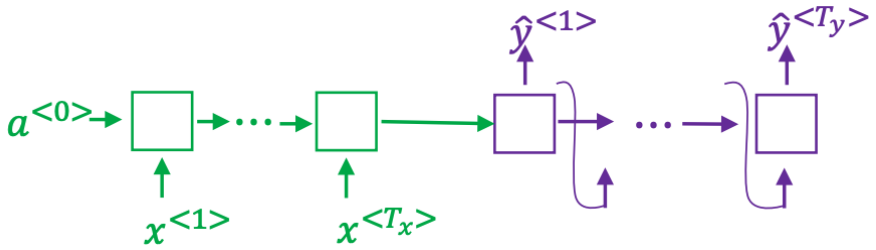

Figure 30: Machine translation, where T_x and T_y an be different.

Machine translation as building a conditional language model

- Language model: $p(y^{\langle 1\rangle}, \cdots, y^{\langle T_y\rangle})$
- Machine translation: encoder $\rightarrow$ Language model
  - "Conditional language model", $p(y^{\langle 1\rangle}, \cdots, y^{\langle T_y\rangle} | x^{\langle 1\rangle}, \cdots, x^{\langle T_x\rangle})$
  - Pick the most likely sentence (translation):
    $\underset{y^{\langle 1\rangle}, \cdots, y^{\langle T_y\rangle}}{\text{argmin}}\ p(y^{\langle 1\rangle}, \cdots, y^{\langle T_y\rangle} | x^{\langle 1\rangle}, \cdots, x^{\langle T_x\rangle})$

15.1 Beam search algorithm

For instance, beam width B = 3, three copies of the network to evaluate the partial sentence fragments and the output are instantiated.

- Step 1: $p(y^{\langle 1\rangle}|x) \rightarrow \hat{y}^{\langle 1\rangle}$
- Step 2: $p(y^{\langle 1\rangle}, y^{\langle 2\rangle}|x) = p(\hat{y}^{\langle 1\rangle}|x)p(\hat{y}^{\langle 2\rangle}|x, \hat{y}^{\langle 1\rangle}) \rightarrow \hat{y}^{\langle 2\rangle}$
- Step 3: $p(y^{\langle 3\rangle}|x, \hat{y}^{\langle 1\rangle}, \hat{y}^{\langle 2\rangle}) \rightarrow \hat{y}^{\langle 3\rangle}$
- $\cdots$

B=1: greedy search

15.1.1 Length normalization

$$\underset{y}{\text{argmin}} \prod_{t=1}^{T_y} p(y^{\langle t \rangle}|x, y^{\langle 1 \rangle}, \cdots, y^{\langle t-1 \rangle}) = \underset{y^{\langle 1 \rangle}, \cdots, y^{\langle T_y \rangle}}{\text{argmin}} p(y^{\langle 1 \rangle}, \cdots, y^{\langle T_y \rangle}|x^{\langle 1 \rangle}, \cdots, x^{\langle T_x \rangle})$$

$$\underset{y}{\text{argmin}} \sum_{t=1}^{T_y} \log p(y^{\langle t \rangle}|x, y^{\langle 1 \rangle}, \cdots, y^{\langle t-1 \rangle})$$

$$\frac{1}{T_y^{\alpha}} \underset{y}{\text{argmin}} \sum_{t=1}^{T_y} \log p(y^{\langle t \rangle}|x, y^{\langle 1 \rangle}, \cdots, y^{\langle t-1 \rangle})$$

hyper-parameter $0 \leq \alpha \leq 1$ for between full and no normalizing by length (hack).

How to choose the beam width B?
Large B: Better result, slower
Small B: Worse result, faster.

Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find exact maximum for $\underset{y}{\text{argmin}} p(y^{\langle t \rangle}|x)$

15.1.2 Error analysis

To figure out what fraction of errors are due to RNN encoder-decoder vs. beam search
Example: $y*$ human translation.

- Case 1
  $p(\hat{y}^{\langle t \rangle}|x) \leq p(y^{*\langle t \rangle}|x)$
  Beam search chose $\hat{y}$, but $y*$ attains higher $p(y|x)$.
  Conclusion: Beam search is fault.

- Case 2
  $p(\hat{y}^{\langle t \rangle}|x) \geq p(y^{*\langle t \rangle}|x)$
  $y*$ is a better translation than $\hat{y}$, but RNN predicted $p(\hat{y}^{\langle t \rangle}|x) > p(y^{*\langle t \rangle}|x)$.
  Conclusion: RNN search is fault.

15.2 Bleu score

Evaluating machine translation
Papineni et. al., 2002. Bleu: A method for automatic evaluation of machine translation
Bleu score on unigrams/bigrams/ngrams
p_n = Bleu score on n-grams only

$$p_n = \frac{\sum_{\text{n-grams} \in \hat{y}} \text{Count}_{\text{Clip}}(\text{n-gram})}{\sum_{\text{n-grams} \in \hat{y}} \text{Count}(\text{n-gram})} \tag{87}$$

Combined Bleu score: $\text{BP}\exp(\frac{1}{4}\sum_{n=1}^{4} p_n)$

BP = brevity penalty

$$BP = \begin{cases} 1, \text{ if MT output length} > \text{reference output length} \\ \exp(1\text{- reference output length / MT output length}), \text{otherwise} \end{cases}$$

15.3 Attention model

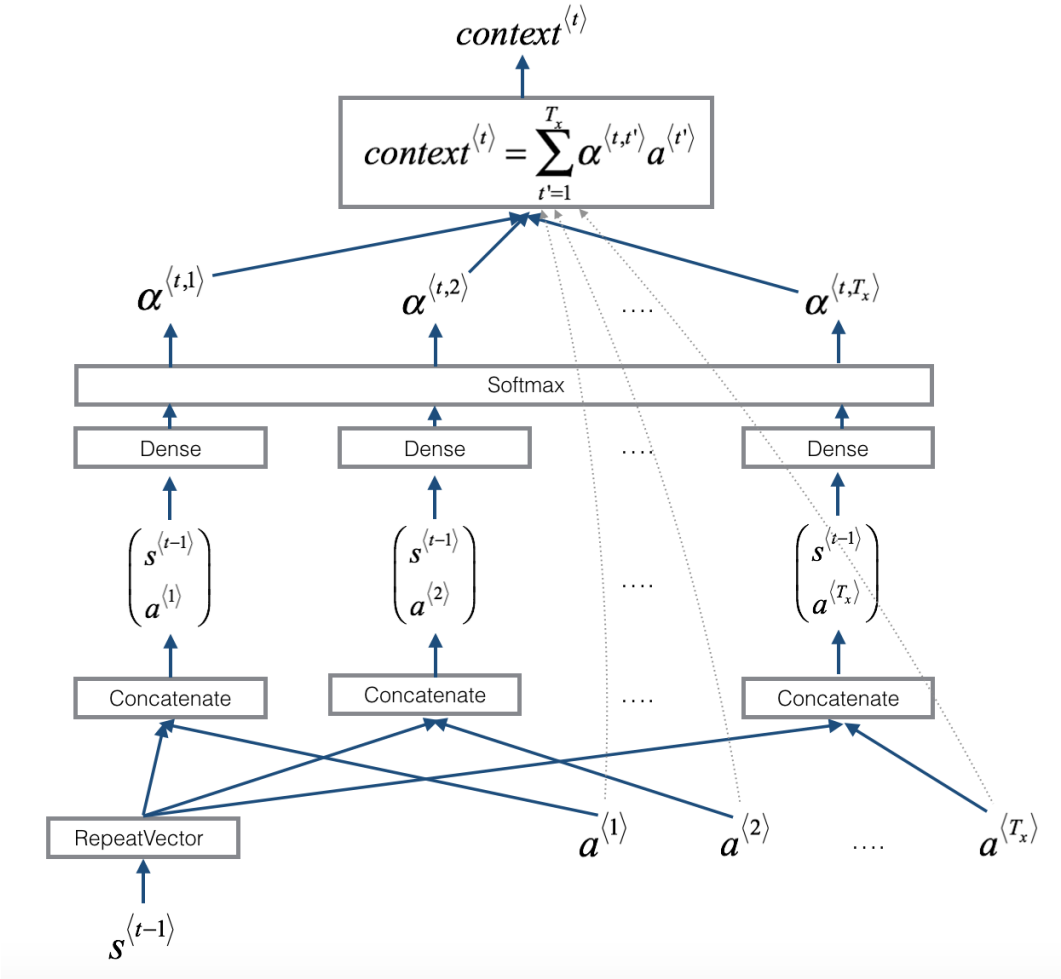

Figure 31: Attention mechanism. RepeatVector node is used to to copy $s^{\langle t \rangle}$'s value T_x times. The concatenation of $s^{\langle t-1 \rangle}$ and $a^{\langle t \rangle}$ is fed into a "Dense" layer, which computes $e^{\langle t,t' \rangle}$. $e^{\langle t,t' \rangle}$ is then passed through a softmax to compute $\alpha^{\langle t,t' \rangle}$.

Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate
Xu et. al., 2015. Show, attend and tell: Neural image caption generation with visual attention

The attention model allows a neural network to pay attention to only part of an input sentence while it's generating a translation. The attention mechanism tells a Neural Machine Translation model where it should pay attention to at any step.

The pre-attention Bi-LSTM goes through T_x time steps. The post-attention LSTM goes through T_y time steps. Note that an LSTM passes both the hidden state $s^{\langle t \rangle}$ and cell state $c^{\langle t \rangle}$ from one time step to the next.

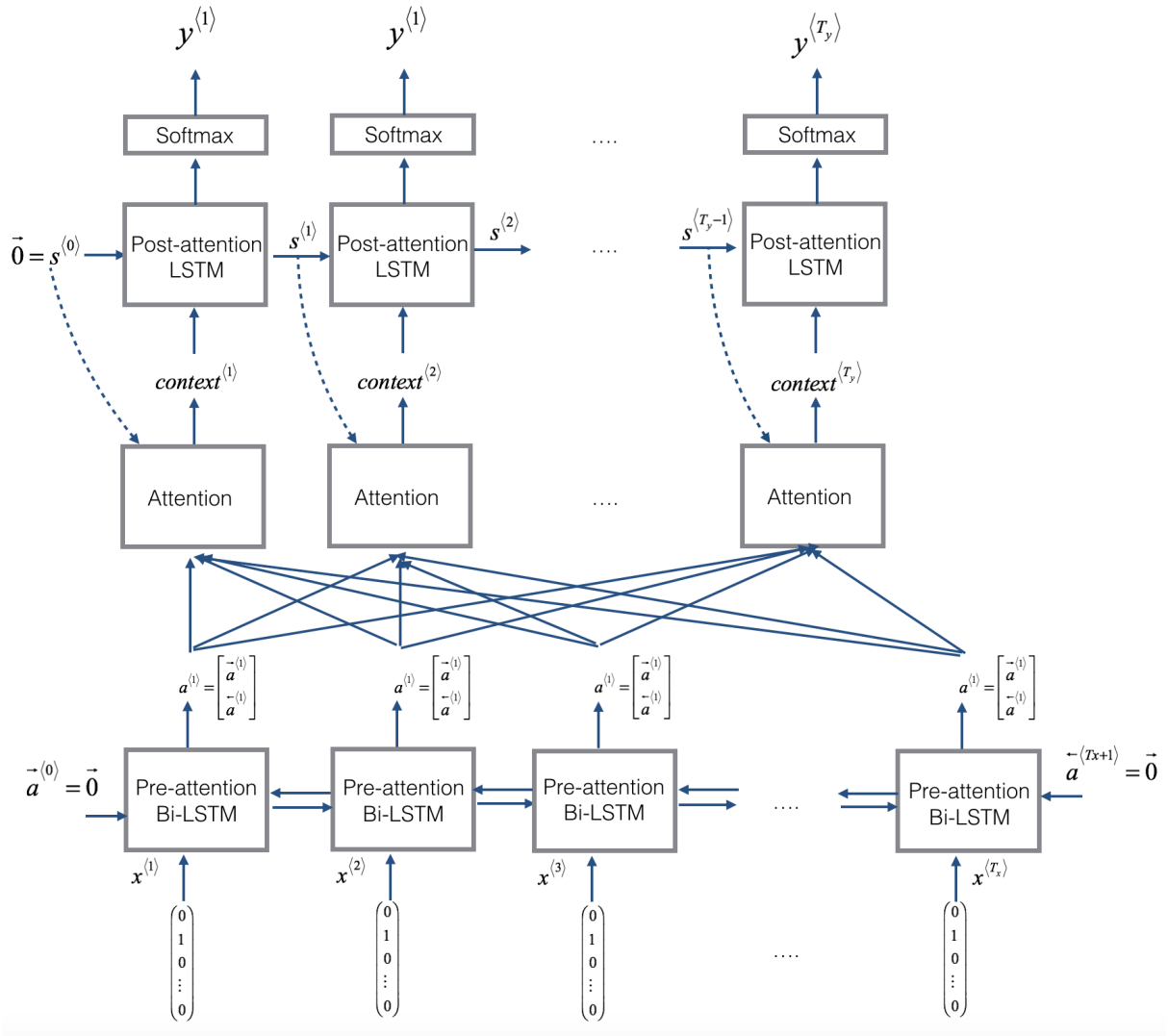

Figure 32: Attention model

$$a^{\langle t' \rangle} = (\overrightarrow{a}^{\langle t' \rangle}, \overleftarrow{a}^{\langle t' \rangle}) \tag{88}$$

$$\sum_{t'} \alpha^{\langle 1,t' \rangle} = 1 \tag{89}$$

$$context^{\langle t \rangle} = \sum_{t'} \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle} \tag{90}$$

$\alpha^{\langle t,t' \rangle}$ = amount of "attention" $y^{\langle t \rangle}$ should pay to $a^{\langle t' \rangle}$

Computing attention $\alpha^{\langle t,t' \rangle}$:

$$\alpha^{\langle t,t' \rangle} = \frac{\exp e^{\langle t,t' \rangle}}{\sum_{t'=1}^{T_x} \exp e^{\langle t,t' \rangle}} \tag{91}$$

attention weights sum to 1, summation over t'.
use a small neural network to compute the "energies" $e^{\langle t,t' \rangle}$ as a function of $s^{\langle t-1 \rangle}$ and $a^{\langle t' \rangle}$. $s^{\langle t-1 \rangle}$ is the hidden state of the post-attention LSTM. $a^{\langle t' \rangle}$ is the hidden state of the pre-attention LSTM.

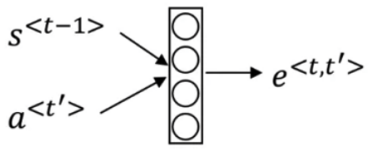

Figure 33: Computing the $e^{\langle t,t' \rangle}$ using a small neural network

Visualization of $\alpha^{\langle t,t' \rangle}$ as a full attention map, to see what the network is paying attention to while generating each output.

15.4 Speech recognition

x (audio clip) $\longrightarrow$ y (transcript)
Attention model for speech recognition
CTC cost for speech recognition
Graves et al., 2006. Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks

Trigger word detection
$x\langle t \rangle$ is features of the audio (such as spectrogram features) at time t.
Using a spectrogram and optionally a 1D CONV layer is a common pre-processing step prior to passing audio data to an RNN, GRU or LSTM.

16 Transformer network

Vaswani et al. 2017, Attention Is All You Need
Attention mechanism + CNN style of processing
Parallel processing: A Transformer Network can ingest entire sentences all at the same time.

- Self-Attention
- Multi-Head Attention

16.0.1 Self-attention

Calculate for each word
A(q, k, v) = attention-based vector representation of a word
Given a word, its neighbouring words are used to compute its context by summing up the word values to map the Attention related to that given word.

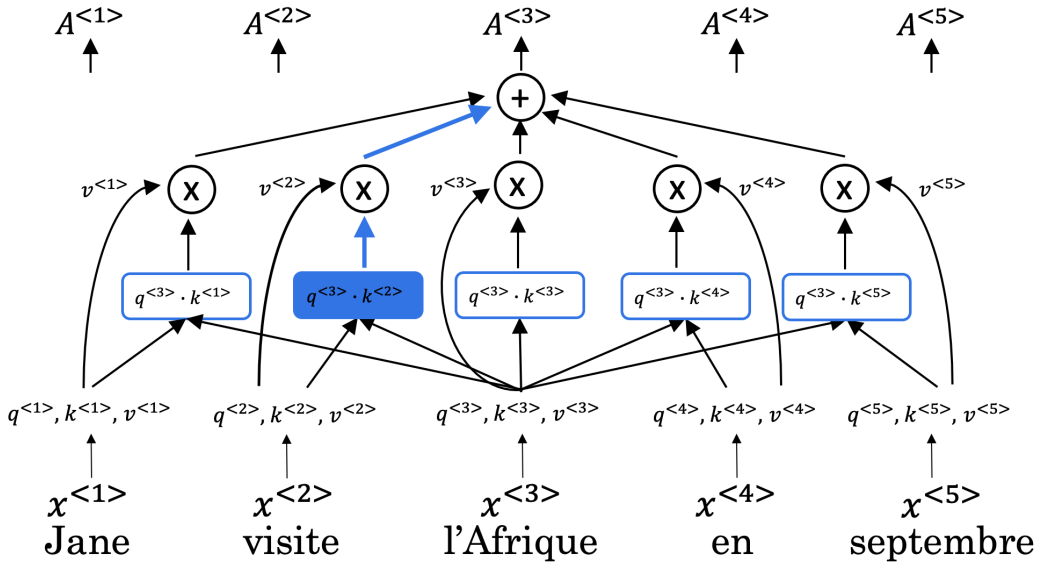

Figure 34: The concept of *Self-Attention*

RNN Attention

$$\alpha^{\langle t,t' \rangle} = \frac{\exp e^{\langle t,t' \rangle}}{\sum_{t'=1}^{T_x} \exp e^{\langle t,t' \rangle}}$$

Transformer Attention

$$A(q,K,V) = \sum_i \frac{\exp e^{\langle q \cdot k^{\langle i \rangle} \rangle}}{\sum_j \exp e^{\langle q \cdot k^{\langle j \rangle} \rangle}} v^{\langle i \rangle}$$

Q = interesting questions about the words in a sentence
K = qualities of words given a Q
V = specific representations of words given a Q

$$\begin{aligned} q^{\langle t \rangle} &= W^Q \cdot x^{\langle t \rangle} \\ k^{\langle t \rangle} &= W^K \cdot x^{\langle t \rangle} \\ v^{\langle t \rangle} &= W^V \cdot x^{\langle t \rangle} \end{aligned}$$

Query (Q)	Key(K)	Value (V)
$q^{\langle 1 \rangle}$	$k^{\langle 1 \rangle}$	$v^{\langle 1 \rangle}$
$q^{\langle 2 \rangle}$	$k^{\langle 2 \rangle}$	$v^{\langle 2 \rangle}$
$q^{\langle 3 \rangle}$	$k^{\langle 3 \rangle}$	$v^{\langle 3 \rangle}$
$q^{\langle 4 \rangle}$	$k^{\langle 4 \rangle}$	$v^{\langle 4 \rangle}$
$q^{\langle 5 \rangle}$	$k^{\langle 5 \rangle}$	$v^{\langle 5 \rangle}$

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{92}$$

d_k is the dimension of the keys, which is used to scale everything down so the softmax doesn't explode.

16.0.2 Multi-head attention

"head": self-attention
Doing the self-attention multiple times to learn a much richer, much better representation for every word.

$$head_i = Attention(W_i^Q Q, W_i^K K, W_i^V V)$$
$$multiHead(Q, K, V) = concat(head_i head_2 \cdots head_h) W_o$$

i here represents the computed attention weight matrix associated with the $i - th$ "head" (sequence).

Transformer
$\langle SOS \rangle x^{\langle 1 \rangle} x^{\langle t2 \rangle} \cdots x^{\langle T_x - 1 \rangle} x^{\langle T_x \rangle} \langle EOS \rangle$ The decoder takes from the Encoder V and K for its second block of Multi-Head Attention, but takes its own cumulative previous predictions Q.

16.0.3 Positional encoding

To encode the positions of the inputs and pass them into the network using sine and cosine formulas:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{93}$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{94}$$

- d the dimension of the word embedding and positional encoding.
- pos is the position of the word.
- i refers to each of the different dimensions of the positional encoding.

The sum of the positional encoding and word embedding is ultimately what is fed into the model. The values of the sine and cosine equations are small enough (between -1 and 1) that when you add the positional encoding to a word embedding, the word embedding is not significantly distorted, and is instead enriched with positional information.

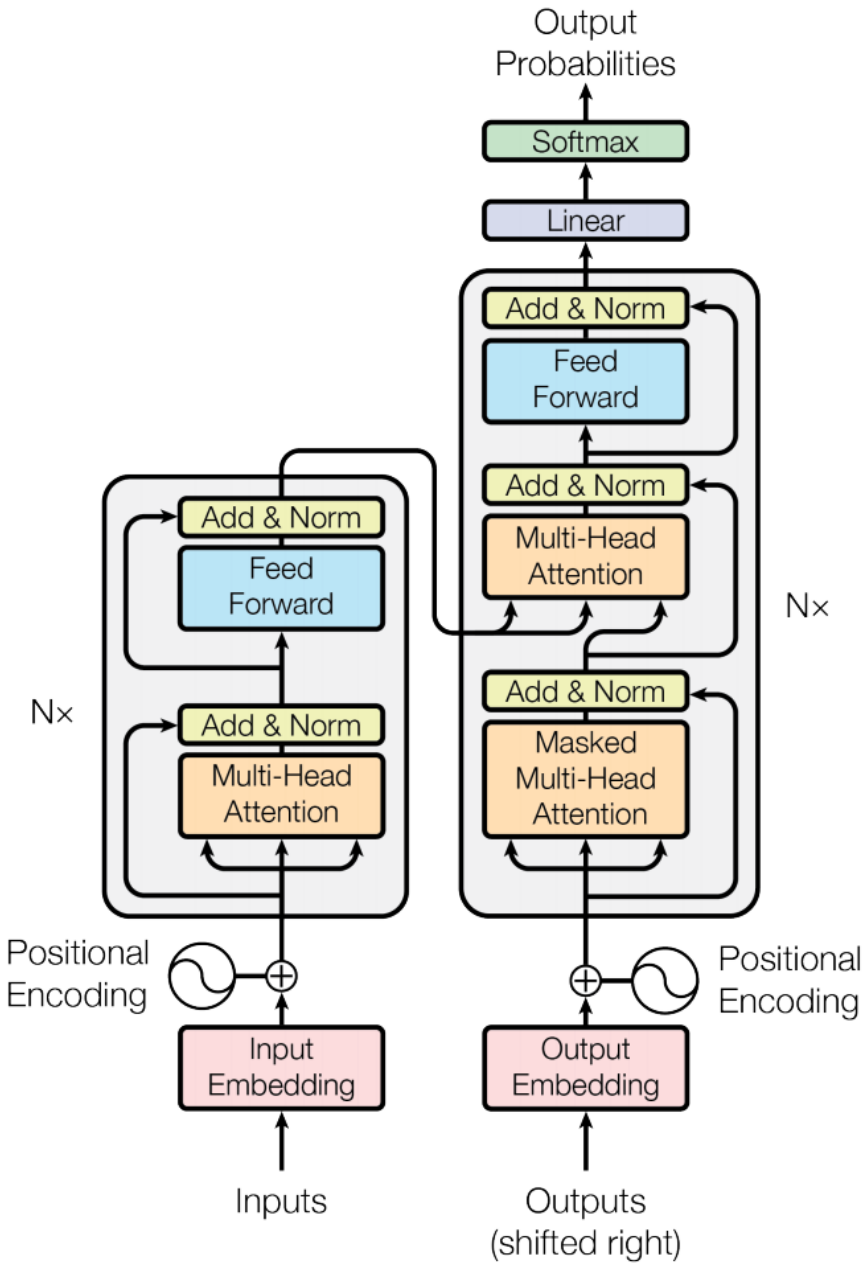

Figure 35: The Transformer - model architecture.