- Introduction
  - Gradient Descent
  - when to use deep learning
  - MMSE
  - Decision Theory
    - MAP
    - ML
  - Entropy
  - Regulatization
  - Train/Test Split
- Simple DNN
  - Forward Propagation
  - Back Propagation
    - idea
    - summary
    - regularizer
  - Multi-Layer Perceptron
    - A Big Picture
    - Activation Function
    - Cost Function
    - Optimizer
  - Regularization
  - Dropout
  - Normalization
  - Dimensionality Reduction
    - PCA
    - LDA
  - Hyper parameter
  - Universal Approximation Theorem
- CNN
  - Introduction
    - Kernel(Filter)
    - Conv layer
    - Pooling layer
    - De-Conv(Transpose convolution) layer
    - Transposed 2D convolution with no padding, stride of 2 and kernel of 3
    - Group Convolution
    - Seperable Convolutions
    - Full Connect layer
  - # of Parameters computation
  - Visualization
  - Different Nets
    - LeNet(1998)

AlexNet(2012)
VGG
Resnet
GoogleNet (Inception)
MobileNet
Comparation
CNN in NLP
Coding
Keras
RNN
Introduction
Vanish Gradient Problem
LSTM
GRU
Back-Propagation through Time (BPTT)
Word Embedding
BERT
TensorFlow and Keras
sequential model
Pytorch


# EE599 Deep Learning

# Introduction

## Gradient Descent

Steepest Gradient Descent

$$\widehat{\mathbf{W}}_{n+1} = \widehat{\mathbf{W}}_n - \eta \nabla_{\mathbf{W}} E(\widehat{\mathbf{W}}_n)$$

Stochastic Gradient Descent

**SGD:** use averages over data (typically subset of data points) to approximate ensemble averaging and therefore approximate the true gradient with a noisy (stochastic) approximation

$$\widehat{\mathbf{W}}_{n+1} = \widehat{\mathbf{W}}_n - \eta \hat{\nabla}_{\mathbf{W}} E(\widehat{\mathbf{W}}_n)$$

single point stochastic gradient descent

## Single Point Stochastic Gradient Descent:

$$
\begin{aligned}
-\frac{1}{2}\nabla_{\mathbf{w}} E &= \mathbf{r}_{xy} - \mathbf{R}_{\mathbf{x}}\mathbf{w} \\
&= \mathbb{E}\left\{y(u)\mathbf{x}(u) - \mathbf{x}(u)\mathbf{x}^{\mathrm{t}}(u)\mathbf{w}\right\} \\
&\approx y_n\mathbf{x}_n - \mathbf{x}_n\mathbf{x}_n^{\mathrm{t}}\mathbf{w} \\
&= (y_n - \mathbf{x}_n^{\mathrm{t}}\hat{\mathbf{w}}_n)\mathbf{x}_n
\end{aligned}
$$

$$
\boxed{\hat{\mathbf{w}}_{n+1} = \hat{\mathbf{w}}_n + \eta(y_n - \hat{\mathbf{w}}_n^t\mathbf{x}_n)\mathbf{x}_n}
$$

# when to use deep learning

Don't

- You have a good statistical model for the inference problem and you can derive and implement the ideal inference function **f**(.)
- A simpler form of supervised ML performs well
  - eg., Linear regression, logistical regression, Naive Bayes Classifier, etc

Do

- Tough to model accurately
- You can use a classical approach, but there are many "clamps" in your algorithm (conditional statements… hacking)
- Modeling is good, but implementing **f**(.) exactly is prohibitively complex

# MMSE

## Minimum Mean-Square Error Estimation (MMSE)

estimate **y**(u) from **x**(u)=**x**

"cross covariance matrix"

$$\mathbf{K}_{\mathbf{yx}} = \mathbb{E}\left\{(\mathbf{y}(u) - \mathbf{m}_{\mathbf{y}})(\mathbf{x}(u) - \mathbf{m}_{\mathbf{x}})^{\mathrm{t}}\right\} = [\mathbf{K}_{\mathbf{xy}}]^{\mathrm{t}}$$

Affine MMSE:

$$\min_{\mathbf{f}(\mathbf{x})=\mathbf{F}\mathbf{x}+\mathbf{b}} \mathbb{E}\left\{\|\mathbf{y}(u) - \mathbf{f}(\mathbf{x}(u))\|^2\right\} \qquad \mathbf{F}_{\mathrm{AMMSE}} = \mathbf{K}_{\mathbf{yx}}\mathbf{K}_{\mathbf{x}}^{-1} \qquad \mathbf{b}_{\mathrm{AMMSE}} = \mathbf{m}_{\mathbf{y}} - \mathbf{F}_{\mathrm{AMMSE}}\mathbf{m}_{\mathbf{x}}$$

$$\min_{\mathbf{F},\mathbf{b}} \mathbb{E}\left\{\|\mathbf{y}(u) - [\mathbf{F}\mathbf{x}(u) + \mathbf{b}]\|^2\right\} \qquad \hat{\mathbf{y}} = \mathbf{K}_{\mathbf{yx}}\mathbf{K}_{\mathbf{x}}^{-1}(\mathbf{x} - \mathbf{m}_{\mathbf{x}}) + \mathbf{m}_{\mathbf{y}} \qquad \mathrm{AMMSE} = \mathrm{tr}\left(\mathbf{K}_{\mathbf{y}} - \mathbf{K}_{\mathbf{yx}}\mathbf{K}_{\mathbf{x}}^{-1}\mathbf{K}_{\mathbf{xy}}\right)$$

Linear MMSE:

$$\min_{\mathbf{f}(\mathbf{x})=\mathbf{F}\mathbf{x}} \mathbb{E}\left\{\|\mathbf{y}(u) - \mathbf{f}(\mathbf{x}(u))\|^2\right\} \qquad \mathbf{F}_{\mathrm{LMMSE}} = \mathbf{R}_{\mathbf{xy}}\mathbf{R}_{\mathbf{x}}^{-1}$$

$$\min_{\mathbf{F}} \mathbb{E}\left\{\|\mathbf{y}(u) - \mathbf{F}\mathbf{x}(u)\|^2\right\} \qquad \hat{\mathbf{y}} = \mathbf{R}_{\mathbf{yx}}\mathbf{R}_{\mathbf{x}}^{-1}\mathbf{x} \qquad \mathrm{LMMSE} = \mathrm{tr}\left(\mathbf{R}_{\mathbf{y}} - \mathbf{R}_{\mathbf{yx}}\mathbf{R}_{\mathbf{x}}^{-1}\mathbf{R}_{\mathbf{xy}}\right)$$

Linear MMSE (typ. means are zero):

$$\hat{y} = \mathbf{w}^{\mathrm{t}}\mathbf{x}$$

$$\mathbf{w} = \mathbf{F}^{\mathrm{t}} = \mathbf{R}_{\mathbf{x}}^{-1}\mathbf{r}_{\mathbf{x}y}$$

$$\mathbf{r}_{\mathbf{x}y} = \mathbf{R}_{\mathbf{x}\mathbf{y}} = \mathbb{E}\left\{\mathbf{x}(u)y(u)\right\} = \left[\mathbb{E}\left\{y(u)\mathbf{x}^{\mathrm{t}}(u)\right\}\right]^{\mathrm{t}}$$

$$\mathrm{LMMSE} = \sigma_y^2 - \mathbf{r}_{\mathbf{x}y}^{\mathrm{t}}\mathbf{R}_{\mathbf{x}}^{-1}\mathbf{r}_{\mathbf{x}y}$$

# Decision Theory

- **Bayesian Decision Theory**
  - Bayes decision rule
  - MAP rule - minimum error probability rule
- Maximum Likelihood
  - Likelihood, Negative-Log-Likelihood, Likelihood ratios
- Neyman-Pearson test and the ROC
  - Detection and False Alarm trade off

Bayes risk for decision rule r

$$\mathrm{Risk}(r) = \int_{\mathcal{Z}} p(\mathbf{z}) \left[\sum_j r(A_j|\mathbf{z})C(A_j|\mathbf{z})\right] d\mathbf{z}$$

Cost for taking action m given observation **z**

$$C(A_j|\mathbf{z}) = \sum_i C(\mathcal{H}_i, A_j)p(\mathcal{H}_i|\mathbf{z})$$

$$C_{i,j} = C(\mathcal{H}_i, A_j) = \text{Cost of deciding } \mathcal{H}_j \text{ when } \mathcal{H}_i \text{ is true}$$

Bayes decision rule (minimizes Bayes risk)

$$r_{\mathrm{bayes}}(A_m|\mathbf{z}) = \begin{cases} 1 & m = \arg\min_j C(A_j|\mathbf{z}) \\ 0 & \text{else} \end{cases}$$

APP factoring

$$p(\mathcal{H}_m|\mathbf{z}) = \frac{p(\mathbf{z}|\mathcal{H}_m)\pi_m}{p(\mathbf{z})}$$

a posteriori probability (APP)

$$\equiv p(\mathbf{z}|\mathcal{H}_m)\pi_m$$

equivalent for making decisions ( p(**z**) not dependent on hypothesis)

## MAP

## MAP Estimation (parameter estimation)

Theta is a parameter set with a known/modeled a priori distribution:
like ML, but with a prior distribution on Theta

$$\hat{\Theta}_{MAP} = \hat{\Theta}_{MAP}(y) = \arg\max_{\theta} p_{y(u)|\Theta(u)}(y|\theta)\, p_{\Theta(u)}(\theta)$$

May be based on a conditional or joint distribution:

$$\hat{\Theta}_{MAP} = \hat{\Theta}_{MAP}(y, \mathbf{x}) = \arg\max_{\theta} p_{y(u)|\mathbf{x}(u),\Theta(u)}(y|\mathbf{x}, \theta)\, p_{\Theta(u)|\mathbf{x}(u)}(\theta|\mathbf{x})$$

$$\hat{\Theta}_{MAP} = \hat{\Theta}_{MAP}(y, \mathbf{x}) = \arg\max_{\theta} p_{y(u),\mathbf{x}(u),\Theta(u)}(y, \mathbf{x}, \theta)\, p_{\Theta(u)}(\theta)$$

# ML

## Maximum Likelihood Parameter Estimation

Theta is a (non-random) parameter set

$$\hat{\Theta}_{ML} = \hat{\Theta}_{ML}(y) = \arg\max_{\Theta} p_{y(u)}(y; \Theta)$$

Alternate notation: $p_{y(u)}(y|\Theta)$ **recall:** likelihood of Theta (given y)

The ML estimate is the parameters values that best explain the observation y under the statistical model for y(u)

May be based on a conditional or joint distribution:

$$\hat{\Theta}_{ML} = \hat{\Theta}_{ML}(y, \mathbf{x}) = \arg\max_{\Theta} p_{y(u)|\mathbf{x}(u)}(y|\mathbf{x}; \Theta)$$

$$\hat{\Theta}_{ML} = \hat{\Theta}_{ML}(y, \mathbf{x}) = \arg\max_{\Theta} p_{\mathbf{x}(u),y(u)}(\mathbf{x}, y; \Theta)$$

## Properties of ML Estimators

- Asymptotically Gaussian:
  - For large amounts of data, the ML estimate is Gaussian with mean equal to the true parameter (models matched)
- Consistent:
  - The limit in probability of the ML estimate is the true parameter (model matched)
- The ML estimate minimizes the KL Divergence between the model distribution and the empirical data distribution. KL divergence measures the difference between two distribution (Info. Theory).
  - Minimizing KL divergence in this case also corresponds to minimizing the cross entropy

# Entropy

Entropy:

$$\begin{aligned} H(X(u)) &= \mathbb{E}\left\{\log_2\left(\frac{1}{p_{X(u)}(X(u))}\right)\right\} \\ &= \sum_k p_{X(u)}(k)\log_2\left(\frac{1}{p_{X(u)}(k)}\right) \\ &= \sum_k p_k \log_2\left(\frac{1}{p_k}\right) \end{aligned}$$

Intuition:

events with low probability have large information
— e.g., "it will snow in Phoenix tomorrow"

the entropy is the average information learned when the value of X(u) is revealed.

weather report in Phoenix has low entropy (almost always the same), whereas in Sioux City, SD it has high entropy (highly variant weather)

Examples:

fair die:

$$H(X(u)) = \log_2(1/6) = 2.58 \quad \text{bits/roll}$$

loaded die:

$$\begin{aligned} H(X(u)) &= -0.4\log_2(0.4) - 0.1\log_2(0.1) - 0.01\log_2(0.01) \\ &\quad - 0.09\log_2(0.09) - 0.25\log_2(0.25) - 0.15\log_2(0.15) \\ &= 2.15 \quad \text{bits/roll} \end{aligned}$$

**cross entropy**

Cross-Entropy

$$\mathrm{CE}(p,\tilde{p}) = \mathbb{E}_p\left\{\log\left(\frac{1}{\tilde{p}_x(X(u))}\right)\right\}$$

**binary cross entropy**

Define the binary cross entropy function:

$$\mathrm{BCE}(p,\tilde{p}) \triangleq \mathbb{E}_p\left\{\log\left(\frac{1}{\tilde{p}(y)}\right)\right\}$$

$$= p\log\frac{1}{\tilde{p}} + (1-p)\log\frac{1}{(1-\tilde{p})}$$

$$= -\left[p\log(\tilde{p}) + (1-p)\log(1-\tilde{p})\right]$$

Can show that minimizing BCE minimizes a measure of difference between p and p-tilde

# Regulatization

enforce penalty on weights to bias toward a prior distribution. --> to reduce over-fitting (by smaller weight)

## Regularization Interpretation…

$$\max_{\theta} p_{y(u)|\mathbf{x}(u),\Theta(u)}(y|\mathbf{x},\theta)\, p_{\Theta(u)|\mathbf{x}(u)}(\theta|\mathbf{x}) \quad \Longleftrightarrow \quad \min_{\mathbf{w}} \|\mathbf{y}-\mathbf{X}\mathbf{w}\|^2 + \lambda\|\mathbf{w}\|^2$$

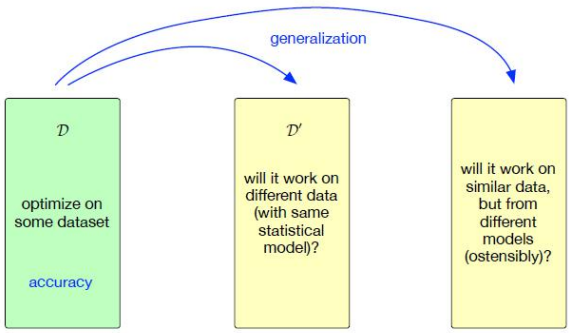
The a-priori Gaussian distribution on the weights leads to "L2 regularization"

penalizes large **w** — even if large w cause smaller squared error

this can be viewed a method to combat **over-fitting**

*lamba* is called the regularization coefficient in this context

*Larger lambda ==> penalize larger weights more aggressively (at expense of SE)*




in the end, we do not really care about the performance on the dataset we have (it is labeled, after all)

we care about performance on similar data that has no labels

Accuracy/Generalization trade-off (aka bias-variance trade):

generally, optimizing accuracy to the extreme will cause reduced generalization capability

## Train/Test Split

Training Set -- for define trainable parameters
Validation Set -- for define hyper-parameters
Test Set -- for verify model performance
Mini- Batch -- for do one SGD update per mini-batch
epoch -- one training run through all data set
iteration -- number of mini-batches per epoch

# Simple DNN

Input an example from a dataset.

The network will take that example and apply some complex computations to it using randomly initialised variables (called weights and biases).

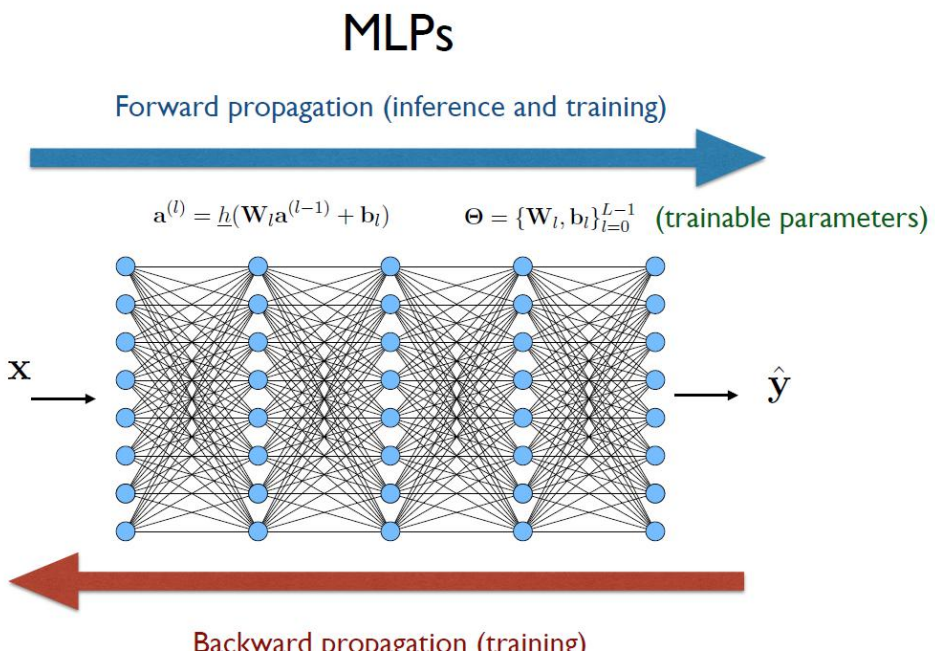
A predicted result will be produced.

Comparing that result to the expected value will give us an error.

Propagating the error back through the same path will adjust the variables.

Steps 1–5 are repeated until we are confident to say that our variables are well-defined.

A predication is made by applying these variables to a new unseen input.




# Forward Propagation

# Back Propagation

## idea

do SGD on all trainable parameters:

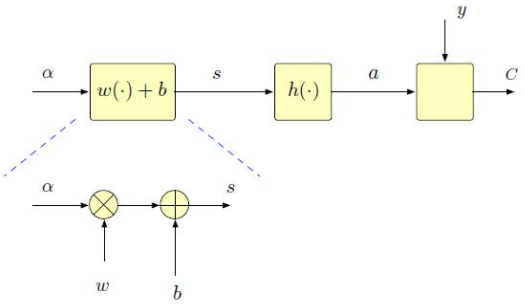
weight updates: $w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial C}{\partial w_{i,j}^{(l)}}$

bias updates: $b_i^{(l)} \leftarrow b_i^{(l)} - \eta \frac{\partial C}{\partial b_i^{(l)}}$

all layers, all indices: $\forall\, l \in \{1, 2, \ldots L\}, i, j$

Backprop is an algorithm for computing these, starting at the neural network output and propagating backward to the input layer using the **chain rule**

To be specified, one step of forward neural net can be extracted as follow:




want to compute: $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$

Note that: $\frac{\partial s}{\partial w} = \alpha \qquad \frac{\partial s}{\partial b} = 1$

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial s}\frac{\partial s}{\partial w} = \alpha\frac{\partial C}{\partial s}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial s}\frac{\partial s}{\partial b} = \frac{\partial C}{\partial s}$$

so we can find $\frac{\partial C}{\partial s}$ and convert to the desired partials easily

$$\frac{\partial C}{\partial s} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial s}$$
$$= \frac{\partial C}{\partial a}\frac{\partial h(s)}{\partial s}$$
$$= \dot{C}(a)\dot{h}(s)$$

shorthand

$$\dot{C}(a) = \frac{\partial C}{\partial a}$$

$$\dot{h}(s) = \frac{dh(s)}{ds}$$

$$\frac{\partial C}{\partial w} = \dot{C}(a)\dot{h}(s)\alpha$$

$$\frac{\partial C}{\partial b} = \dot{C}(a)\dot{h}(s)$$

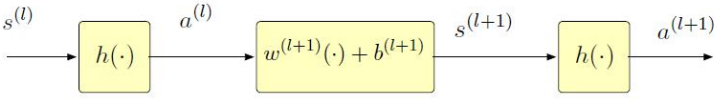
we know how to compute: $\frac{\partial C}{\partial s}$

and how to convert to:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial s}\frac{\partial s}{\partial w} = \alpha\frac{\partial C}{\partial s}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial s}\frac{\partial s}{\partial b} = \frac{\partial C}{\partial s}$$

now consider:



Shorthand: $\delta^{(l)} \triangleq \frac{\partial C}{\partial s^{(l)}}$

$$\frac{\partial C}{\partial s^{(l)}} \leftarrow \frac{\partial C}{\partial s^{(l+1)}}$$

$$\boxed{\delta^{(l)} \leftarrow \delta^{(l+1)}}$$

$$\frac{\partial C}{\partial w^{(l)}} = \frac{\partial C}{\partial s^{(l)}} \frac{\partial s^{(l)}}{\partial w^{(l)}} = \frac{\partial C}{\partial s^{(l)}} a^{(l-1)}$$

$$\frac{\partial C}{\partial w^{(l)}} = \delta^{(l)} a^{(l-1)}$$

$$\frac{\partial C}{\partial b^{(l)}} = \frac{\partial C}{\partial s^{(l)}} \frac{\partial s^{(l)}}{\partial b^{(l)}} = \frac{\partial C}{\partial s^{(l)}}$$

$$\frac{\partial C}{\partial b^{(l)}} = \delta^{(l)}$$

$$\boxed{\begin{aligned} w^{(l)} &\leftarrow w^{(l)} - \eta \delta^{(l)} a^{(l-1)} \\ b^{(l)} &\leftarrow b^{(l)} - \eta \delta^{(l)} \end{aligned}}$$

© Keith M. Chugg, 2019



$$\begin{aligned} \delta^{(l)} &= \frac{\partial C}{\partial s^{(l)}} = \frac{\partial C}{\partial s^{(l+1)}} \frac{\partial s^{(l+1)}}{\partial s^{(l)}} \\ &= \delta^{(l+1)} \frac{\partial s^{(l+1)}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial s^{(l)}} \\ &= \delta^{(l+1)} w^{(l+1)} \dot{h}(s^{(l)}) \\ &= \dot{h}(s^{(l)}) w^{(l+1)} \delta^{(l+1)} \\ &= \dot{a}^{(l)} w^{(l+1)} \delta^{(l+1)} \end{aligned}$$

$$\delta^{(l)} \leftarrow \delta^{(l+1)}$$

Shorthand:

$$\dot{a}^{(l)} \triangleq \dot{h}(s^{(l)})$$

$$\boxed{\begin{aligned} w^{(l)} &\leftarrow w^{(l)} - \eta \delta^{(l)} a^{(l-1)} \\ b^{(l)} &\leftarrow b^{(l)} - \eta \delta^{(l)} \end{aligned}}$$

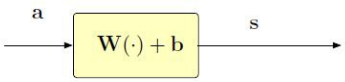
**Note:** you update the weights and biases only after all the deltas are computed

$$\boxed{\begin{aligned} w^{(l)} &\leftarrow w^{(l)} - \eta \delta^{(l)} a^{(l-1)} \\ b^{(l)} &\leftarrow b^{(l)} - \eta \delta^{(l)} \end{aligned}}$$

work for matrix (vector case)

now we just need to handle the vector case…

(let's just repeat what we did)




$$\mathbf{s} = \mathbf{W}\mathbf{a} + \mathbf{b} \qquad \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \qquad s_n = \left( \sum_m w_{nm} a_m \right) + b_n$$

Suppose we know: $\nabla_{\mathbf{s}} C = \boldsymbol{\delta}$

How to convert to: $\dfrac{\partial C}{\partial w_{ij}} \qquad \dfrac{\partial C}{\partial b_i}$

$$\frac{\partial C(s_0(w_0, w_1), s_1(w_0, w_1))}{\partial w_0} = \frac{\partial C(s_0(w_0, w_1), s_1(w_0, w_1))}{\partial s_0} \frac{\partial s_0}{\partial w_0} + \frac{\partial C(s_0(w_0, w_1), s_1(w_0, w_1))}{\partial s_1} \frac{\partial s_1}{\partial w_0}$$

$$\frac{\partial C}{\partial w_0} = \frac{\partial C}{\partial s_0} \frac{\partial s_0}{\partial w_0} + \frac{\partial C}{\partial s_1} \frac{\partial s_1}{\partial w_0}$$

$$\begin{aligned} \delta_i^{(l)} &= \sum_m \delta_m^{(l+1)} \frac{\partial s_m^{(l+1)}}{\partial s_i^{(l)}} \\ &= \sum_m \delta_m^{(l+1)} w_{mi} \dot{h}(s_i^{(l)}) \\ &= \left( \sum_m \delta_m^{(l+1)} w_{mi} \right) \dot{h}(s_i^{(l)}) \\ &= \left( \sum_m \delta_m^{(l+1)} w_{mi} \right) \dot{a}_i^{(l)} \\ &= \left[ \left( \mathbf{W}^{(l+1)} \right)^{\mathrm{t}} \boldsymbol{\delta}^{(l+1)} \right]_i \dot{a}_i^{(l)} \end{aligned}$$

$$\boldsymbol{\delta}^{(l)} = \dot{\mathbf{a}}^{(l)} \odot \left[ \left( \mathbf{W}^{(l+1)} \right)^{\mathrm{t}} \boldsymbol{\delta}^{(l+1)} \right]$$

shorthand:

$$\dot{a}_i^{(l)} = \dot{h}(s_i^{(l)})$$

$$\dot{\mathbf{a}}^{(l)} = \underline{\dot{h}}(\mathbf{s}^{(l)})$$

$$\mathbf{v} \odot \mathbf{w} = \begin{bmatrix} v_0 w_0 \\ v_1 w_1 \\ \vdots \\ v_{D-1} w_{D-1} \end{bmatrix}$$

Hadamard product: *. in matlab or * for np.arrays

**summary**

$$\mathbf{a}^{(l)} = \underline{h}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

$$\dot{\mathbf{a}}^{(l)} = \underline{\dot{h}}\left(\mathbf{W}^{(l)}\mathbf{a}_{i-1} + \mathbf{b}_i\right)$$

$$\boldsymbol{\delta}^{(L)} = \dot{\mathbf{a}}^{(L)} \odot \underline{\dot{C}}\left(\mathbf{y}, \mathbf{a}^{(l)}\right)$$

$$\boldsymbol{\delta}^{(l)} = \dot{\mathbf{a}}^{(l)} \odot \left[\left(\mathbf{W}^{(l+1)}\right)^{\mathrm{t}} \boldsymbol{\delta}^{(l+1)}\right]$$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\boldsymbol{\delta}^{(l)}\left[\mathbf{a}^{(l-1)}\right]^{\mathrm{t}}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta\boldsymbol{\delta}^{(l)}$$

specialized to vectorized output activation and additive cost

$$\mathbf{a}_l = \mathrm{act}(\mathbf{W}_l\mathbf{a}_{l-1} + \mathbf{b}_l) \qquad \text{(activations)}$$

$$\dot{\mathbf{a}}_l = \dot{\mathrm{act}}(\mathbf{W}_l\mathbf{a}_{l-1} + \mathbf{b}_l) \qquad \text{(derivative activations)}$$

$$\boldsymbol{\delta}_L = \dot{\mathbf{a}}_L \odot \dot{\mathrm{cost}}(\mathbf{y}, \mathbf{a}_L) \qquad \text{(delta initialization)}$$

$$\boldsymbol{\delta}_l = \dot{\mathbf{a}}_l \odot \left[\mathbf{W}_{l+1}^{\mathrm{t}}\boldsymbol{\delta}_{l+1}\right] \qquad \text{(delta recursion)}$$

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta\boldsymbol{\delta}_l\mathbf{a}_{l-1}^{\mathrm{t}} \qquad \text{(weight SGD update)}$$

$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta\boldsymbol{\delta}_l \qquad \text{(bias SGD update)}$$

specialized to vectorized output activation and additive cost

**using batch**

so it's like compute several data points with same **w** and **δ** and use average as the update delta

For each data sample: $(\mathbf{x}[n], \mathbf{y}[n])$

Do FF and BP to compute activations, a-dots, and deltas

$$\mathbf{a}_l[n] = \text{act}(\mathbf{W}_l \mathbf{a}_{l-1}[n] + \mathbf{b}_l) \quad \text{(activations)}$$

$$\dot{\mathbf{a}}_l[n] = \dot{\text{act}}(\mathbf{W}_l \mathbf{a}_{l-1}[n] + \mathbf{b}_l) \quad \text{(derivative activations)}$$

$$\boldsymbol{\delta}_L[n] = \dot{\mathbf{a}}_L[n] \odot \dot{\text{cost}}(\mathbf{y}[n], \mathbf{a}_L[n]) \quad \text{(delta initialization)}$$

$$\boldsymbol{\delta}_l[n] = \dot{\mathbf{a}}_l[n] \odot \left[\mathbf{W}_{l+1}^{\text{t}} \boldsymbol{\delta}_{l+1}[n]\right] \quad \text{(delta recursion)}$$

Do one SGD update after finishing the batch

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta \frac{1}{B} \sum_{n=0}^{B-1} \boldsymbol{\delta}_l[n] \mathbf{a}_{l-1}^{\text{t}}[n]$$

$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta \frac{1}{B} \sum_{n=0}^{B-1} \boldsymbol{\delta}_l[n]$$

### regularizer

For typical weight-penalty regularizers (e.g., L1 and L2), these are direct functions of the weights

Example: $$C_{\text{reg}} = C + \lambda \sum_l \|\mathbf{W}^{(l)}\|^2 = C + \lambda \sum_l \sum_{i,j} \left[\mathbf{W}_{i,j}^{(l)}\right]^2$$

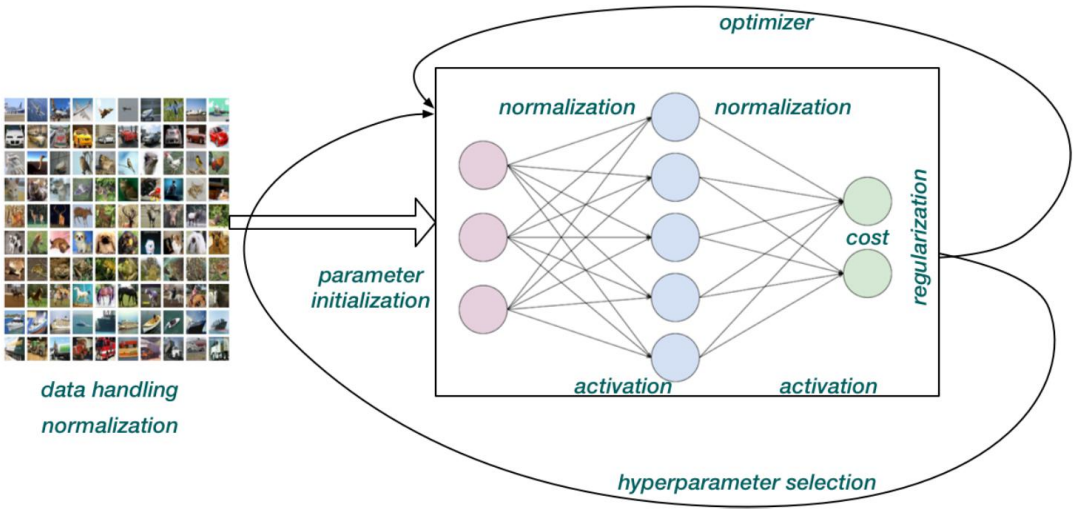
$$\frac{\partial C_{\text{reg}}}{\partial w_{i,j}^{(l)}} = \frac{\partial C}{\partial w_{i,j}^{(l)}} + 2\lambda w_{i,j}^{(l)}$$

Minor change to gradient update:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \left( \boldsymbol{\delta}^{(l)} \left[\mathbf{a}^{(l-1)}\right]^{\text{t}} + 2\lambda \mathbf{W}^{(l)} \right)$$

# Multi-Layer Perceptron

## A Big Picture




## Activation Function

**middle layers Activation Function**

sigmoid & tanh

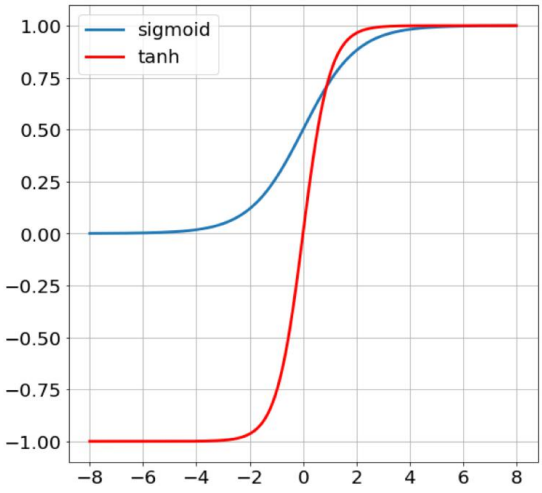
Squashing activations

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = 2\sigma(2x) - 1$$

Hyperbolic Tangent
(just rescaled sigmoid)




problem: vanish

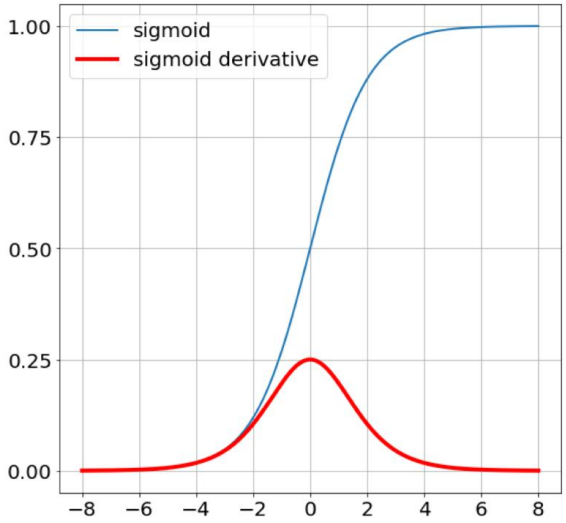
# Vanishing gradients

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$$

Recall from BP (take example L=3)

$$\delta^{(1)} = H'^{(1)} W^{(2)^T} H'^{(2)} W^{(3)^T} H'^{(3)} \nabla_{a^{(3)}} C$$

*All these numbers are <= 0.25, so gradients become very small!!*



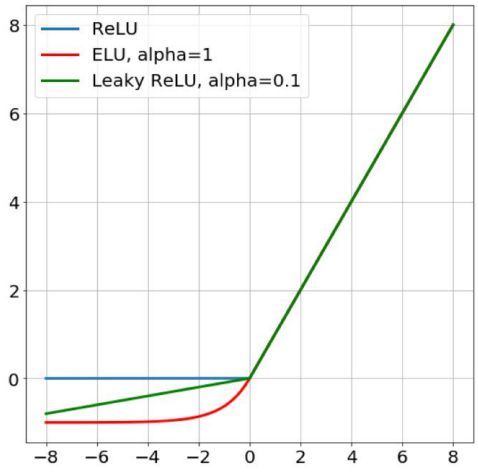

more important: **ReLU family -- will not vanish**

# ReLU family of activations

| | x >= 0 | x < 0 |
|---|---|---|
| Rectified Linear Unit (ReLU) | x | 0 |
| Exponential Linear Unit (ELU) | x | α(e^x - 1) |
| Leaky ReLU | x | αx |

Biologically inspired - *neurons firing vs not firing*
Solves vanishing gradient problem
Non-differentiable at 0, replace with anything in [0,1]

ReLU can die if x<0
Leaky ReLU solves this, but inconsistent results
ELU saturates for x<0, so less resistant to noise




Clevert, Djork-Arné; Unterthiner, Thomas; Hochreiter, Sepp (2015-11-23). "Fast and

# Maxout networks - Generalization of ReLU

Normally:

$$\underset{N^{(l)}\times 1}{s^{(l)}} = \underset{N^{(l)}\times N^{(l-1)}}{W^{(l)}} \underset{N^{(l-1)}\times 1}{a^{(l-1)}} + \underset{N^{(l)}\times 1}{b^{(l)}} \qquad \underset{N^{(l)}\times 1}{a^{(l)}} = h\left(\underset{N^{(l)}\times 1}{s^{(l)}}\right)$$

For maxout:

$$\underset{k\times N^{(l)}\times 1}{s^{(l)}} = \underset{k\times N^{(l)}\times N^{(l-1)}}{W^{(l)}} \underset{N^{(l-1)}\times 1}{a^{(l-1)}} + \underset{k\times N^{(l)}\times 1}{b^{(l)}}$$

$$\underset{N^{(l)}\times 1}{a^{(l)}} = \max_{\text{rows}} \underset{k\times N^{(l)}\times 1}{s^{(l)}}$$

Learns the activation function itself
Better approximation power
Takes more computation

Goodfellow, Ian J.; Warde-Farley, David; Mirza, Mehdi; Courville, Aaron; Bengio, Yoshua

## Example of maxout

k = 2, $N^{(l)} = 5$

$$\underset{2\times 5}{\boldsymbol{s}^{(l)}} = \begin{bmatrix} 2.2 & -1.4 & 9.7 & -1.3 & 0 \\ -4 & -2.1 & 3.8 & -3 & 0.2 \end{bmatrix}$$

$$\boldsymbol{a}^{(l)} = \begin{bmatrix} 2.2 & -1.4 & 9.7 & -1.3 & 0.2 \end{bmatrix}^T$$

*ReLU is a special case of maxout with k=2 and 1 set of (**W**, b) = all 0s*

**Output Activation Function**

## Output layer activation - Softmax

$$\boldsymbol{a} = \begin{bmatrix} \frac{e^{s_1}}{\sum_{i=1}^{N} e^{s_i}} \\ \vdots \\ \frac{e^{s_N}}{\sum_{i=1}^{N} e^{s_i}} \end{bmatrix}$$

Extending logistic regression to multiple classes

*Network output is a probability distribution!*

Compare to ideal output probability distribution

$$\boldsymbol{y}^{(L)} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

## Cost Function

Cross-entropy Cost

One Hot label vs. Softmax

Quadratic Cost (MSE)

$$C = -\sum_{i=1}^{N^{(L)}} y_i^{(L)} \ln a_i^{(L)}$$

Ground truth labels — Network outputs

*Minimizing cross-entropy is the same as minimizing KL-divergence between the probability distributions **y** and **a***

For binary labels, this reduces to:

$$C = -\left[y^{(L)} \ln\left(a^{(L)}\right) + \left(1 - y^{(L)}\right) \ln\left(1 - a^{(L)}\right)\right]$$

## Softmax and cross-entropy with one-hot labels

Recall:

$$\boldsymbol{\delta^{(L)}} = \boldsymbol{H'^{(L)^T}} \boldsymbol{\nabla_{a^{(L)}}} C$$

$$\delta_r^{(L)} = a_r^{(L)} - 1$$
$$\delta_{\neq r}^{(L)} = a_{\neq r}^{(L)}$$

Combining: $\boxed{\boldsymbol{\delta}^{(L)} = \boldsymbol{a}^{(L)} - \boldsymbol{y}^{(L)}}$

*This makes beautiful sense as the error vector!*

## Optimizer

### Hessian

## Hessian: Matrix of double derivatives

Given $f(\boldsymbol{w}) : \mathbb{R}^n \to \mathbb{R}$

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_m \partial w_1} & \cdots & \frac{\partial^2 f}{\partial w_m^2} \end{bmatrix}$$

For continuous functions:

$$\frac{\partial^2 f}{\partial w_i \partial w_j} = \frac{\partial^2 f}{\partial w_j \partial w_i}$$

Hessian is symmetric

### Condition

# Conditioning in terms of Hessian

*Conditioning is measured as the condition number of the Hessian of the cost function*

$$\kappa = \frac{\sigma_{\max}}{\sigma_{\min}} = \frac{|d_{\max}|}{|d_{\min}|}$$

I'm using *d* for eigenvalues

Examples:

$C(w_1, w_2) = (w_1 - 2)^2 + (w_2 - 3)^2$ $\kappa$ = 1 => Well-conditioned

$C(w_1, w_2) = (w_1 - 2)^2 + 9(w_2 - 3)^2$ $\kappa$ = 9 => Ill-conditioned

will condition lead to fast gradient

**Momentum**

Normal update:

$$p \leftarrow p - \eta \nabla_p C$$

Momentum update:

$$v \leftarrow \alpha v - \eta \nabla_p C$$

$$p \leftarrow p + v$$

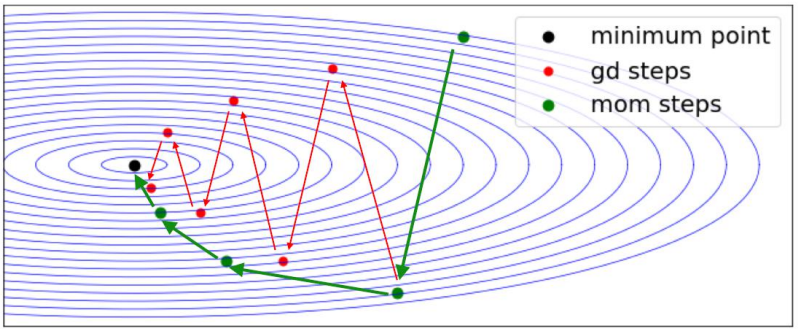
*Equivalent to smoothing the update by a low pass filter*

$\alpha \in [0, 1]$ How much of past history should be applied, typically ~0.9

α (Momentum factor) normally i**ncrease** with the step goes

η (Learning Rate) **decrease** with step goes

Gradient descent with momentum
converges quickly even when ill-conditioned




Nesterov Momentum

# Nesterov Momentum

Normal update:

$$p \leftarrow p - \eta \nabla_p C$$

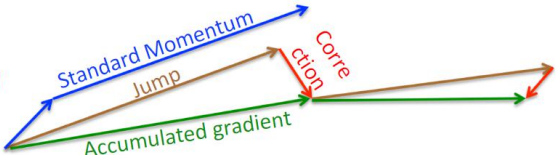
Momentum update:

$$v \leftarrow \alpha v - \eta \nabla_p C$$
$$p \leftarrow p + v$$

Nesterov Momentum update:

$$\tilde{p} \leftarrow p + \alpha v$$
$$v \leftarrow \alpha v - \eta \nabla_{\tilde{p}} C\left(\tilde{p}\right)$$
$$p \leftarrow p + v$$

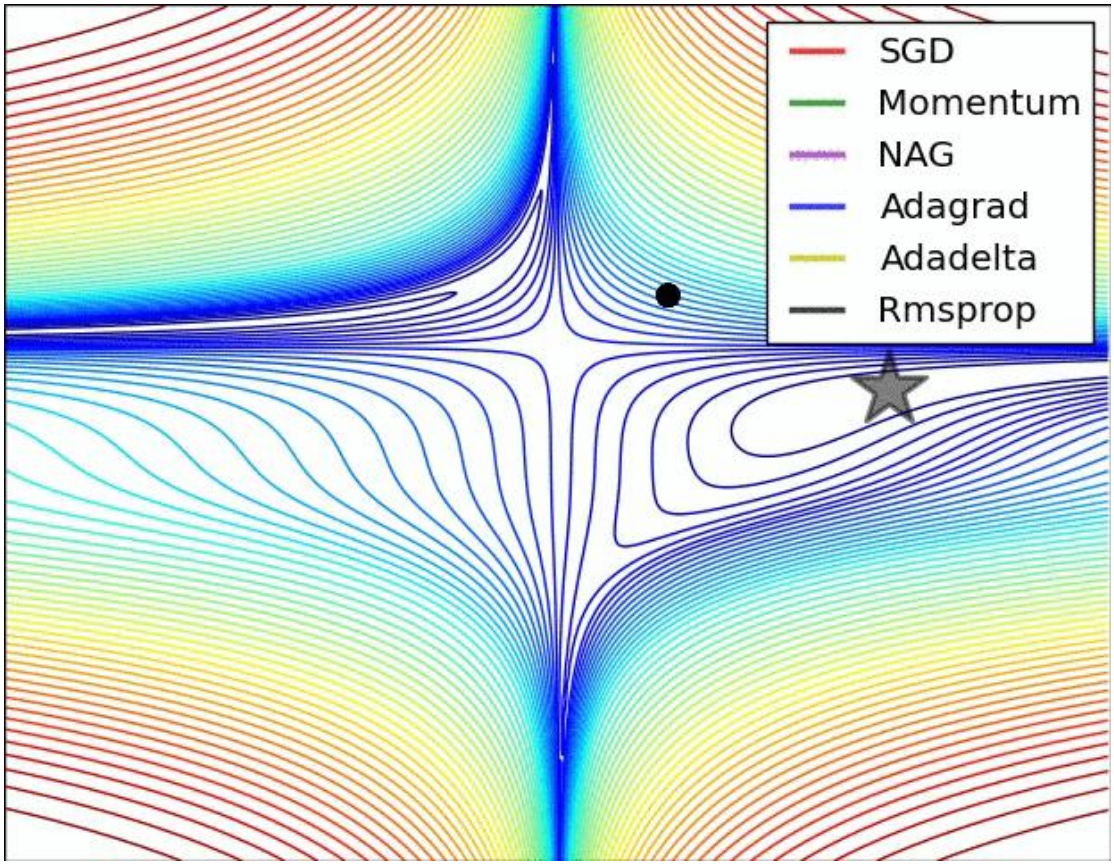
*Correction factor applied to momentum*
*May give faster convergence*

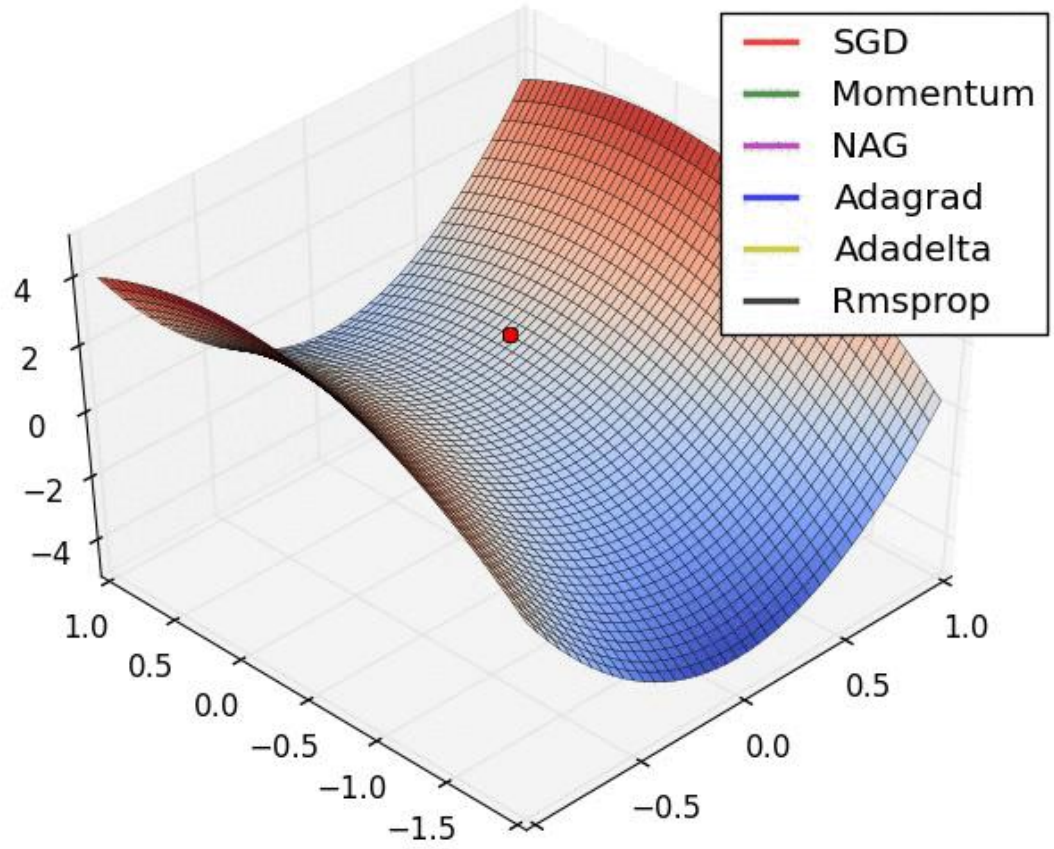



**Differenct Kinds of Optimizers**

# Adaptive Optimizers

- ✓Learning rate should be reduced with time. We don't want to overshoot the minimum point and oscillate.
- ✓Learning rate should be small for sensitive parameters, and vice-versa.
- ✓Momentum factor should be increased with time to get smoother updates.






**RMSprop**

*Scale each gradient by an exponentially weighted moving average of its past history*

$$v \leftarrow \rho v + (1-\rho)\left(\nabla_p C\right)^2$$
$$p \leftarrow p - \frac{\eta}{\sqrt{v}+\epsilon}\nabla_p C$$

Default ρ = 0.9

ϵ is just a small number to prevent division by 0. Can be machine epsilon

**Adam (more complex compared to RMSprop)**

# Adam

*RMSprop with momentum and bias correction*

At time step (t+1):

$$v_1 \leftarrow \rho_1 v_1 + (1-\rho_1)\nabla_p C$$

Momentum

$$v_2 \leftarrow \rho_2 v_2 + (1-\rho_2)\left(\nabla_p C\right)^2$$

Exponentially weighted moving average of past history

$$\tilde{v}_1 = \frac{v_1}{1-\rho_1^t}$$

$$\tilde{v}_2 = \frac{v_2}{1-\rho_2^t}$$

Bias corrections to make

$$\mathbb{E}\{\tilde{v}_1\} = \mathbb{E}\{\nabla_p C\}$$

$$\mathbb{E}\{\tilde{v}_2\} = \mathbb{E}\left\{(\nabla_p C)^2\right\}$$

$$p \leftarrow p - \frac{\eta}{\sqrt{\tilde{v}_2}+\epsilon}\tilde{v}_1$$

Defaults:

η=0.001, $\rho_1$ = 0.9, $\rho_2$ = 0.999

**SGD**

(may include momentum and learning rate)

more basic, need more experience

**AdaMax**

The $v_t$ factor in the Adam update rule scales the gradient inversely proportionally to the $\ell_2$ norm of the past gradients (via the $v_{t-1}$ term) and current gradient $|g_t|^2$:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)|g_t|^2$$

We can generalize this update to the $\ell_p$ norm. Note that Kingma and Ba also parameterize $\beta_2$ as $\beta_2^p$:

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p$$

Norms for large $p$ values generally become numerically unstable, which is why $\ell_1$ and $\ell_2$ norms are most common in practice. However, $\ell_\infty$ also generally exhibits stable behavior. For this reason, the authors propose AdaMax (Kingma and Ba, 2015) and show that $v_t$ with $\ell_\infty$ converges to the following more stable value. To avoid confusion with Adam, we use $u_t$ to denote the infinity norm-constrained $v_t$:

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1-\beta_2^\infty)|g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

We can now plug this into the Adam update equation by replacing $\sqrt{\hat{v}_t} + \epsilon$ with $u_t$ to obtain the AdaMax update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t$$

Note that as $u_t$ relies on the max operation, it is not as suggestible to bias towards zero as $m_t$ and $v_t$ in Adam, which is why we do not need to compute a bias correction for $u_t$ . Good default values are again $\eta = 0.002$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.

**Adadelta (fastest)**

Adadelta [13] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size $w$.

Instead of inefficiently storing $w$ previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step $t$ then depends (as a fraction $\gamma$ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2.$$

We set $\gamma$ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

We now simply replace the diagonal matrix $G_t$ with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2.$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate $\eta$ in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

**which optimizer is best?**

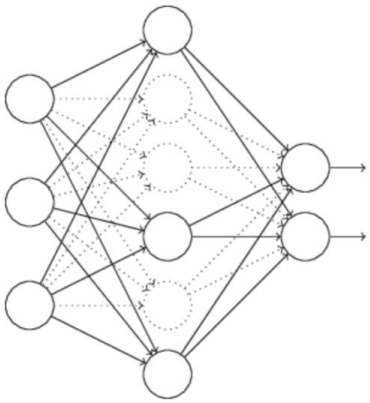
**RMSprop** is an extension of Adagrad that deals with its radically diminishing learning rates. It is **identical to Adadelta**, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, **RMSprop, Adadelta**, and **Adam** are very similar algorithms that do well in similar circumstances.

# Regularization

bias- variance trade-off (in EE660) (**for a given MSE**)

# Dropout

*For every minibatch during training, delete a random selection of input and hidden nodes*



Keep probability $p_k$ : Fraction of units to keep (i.e. not drop)

But why dropout??

Esemble method turn to drop out

## Return to dropout

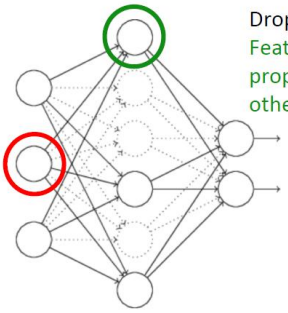
Dropout is an ensemble method using the same shared weights
=> Computationally cheaper

*Improves robustness, since individual nodes have to work without depending on other deleted nodes => Regularization*

## How much dropout?

Original paper recommendations:

Input $p_k = 0.8$ Hidden $p_k = 0.5$ Output $p_k = 1$



Drop hidden:
Features still propagate through other hidden nodes

Drop output: Cannot classify!

0.5 gives maximum regularization effect:
P. Baldi, P. J. Sadowski, "Understanding Dropout", Proc. NIPS 2013

But you should try other values as well!

Conv layer $p_k = 0.7$

Drop input: Feature gone forever!
Some people use input $p_k = 1$

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

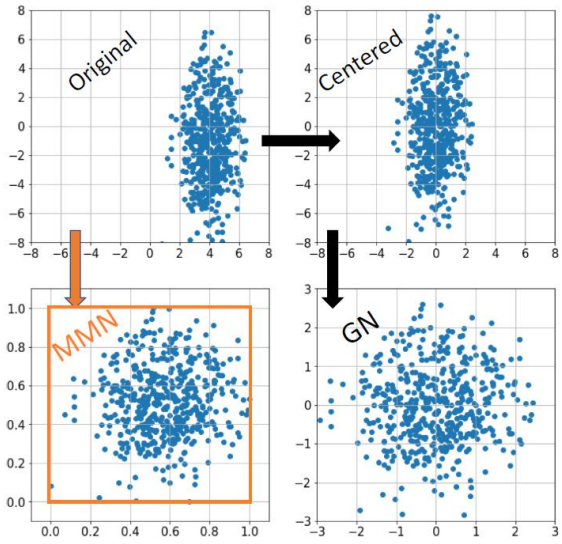
- All nodes and weights are present in inference. Multiply weights by $p_k$ during inference to compensate
  - This assumes linearity, but still works!
- Dropout is like multiplicative 0-1 noise for each node
- Dropout is best for big networks
  - Increase capacity, increase generalization power, but prevent overfitting

# Normalization

## Essential preprocessing

$$\boldsymbol{X}_{:,i} = \frac{\boldsymbol{X}_{:,i} - \mu_i}{\sigma_i}$$

Gaussian normalization:
Each feature is a unit Gaussian




$$\boldsymbol{X}_{:,i} = \frac{\boldsymbol{X}_{:,i} - m_i}{M_i - m_i}$$

Minmax normalization:
Each feature is in [0,1]

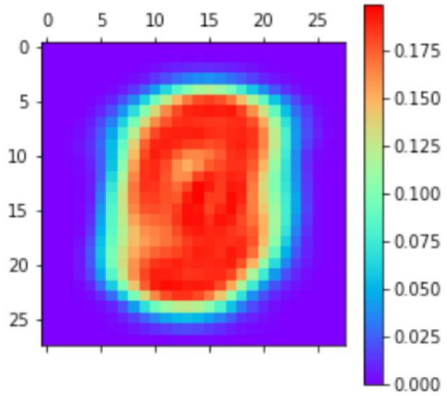
# Dimensionality Reduction

SVD: singular value decomposition

## PCA

### Principal Component Analysis (PCA)

*Keep features with maximum variance*
*=> features which change the most*



Assume sorted

$$\underset{f\times f}{\boldsymbol{U}} \quad \underset{f\times f}{\boldsymbol{\Sigma}} \quad \underset{f\times f}{\boldsymbol{V}} = \mathrm{SVD}\left\{\mathrm{Cov}(\boldsymbol{X})\right\}$$

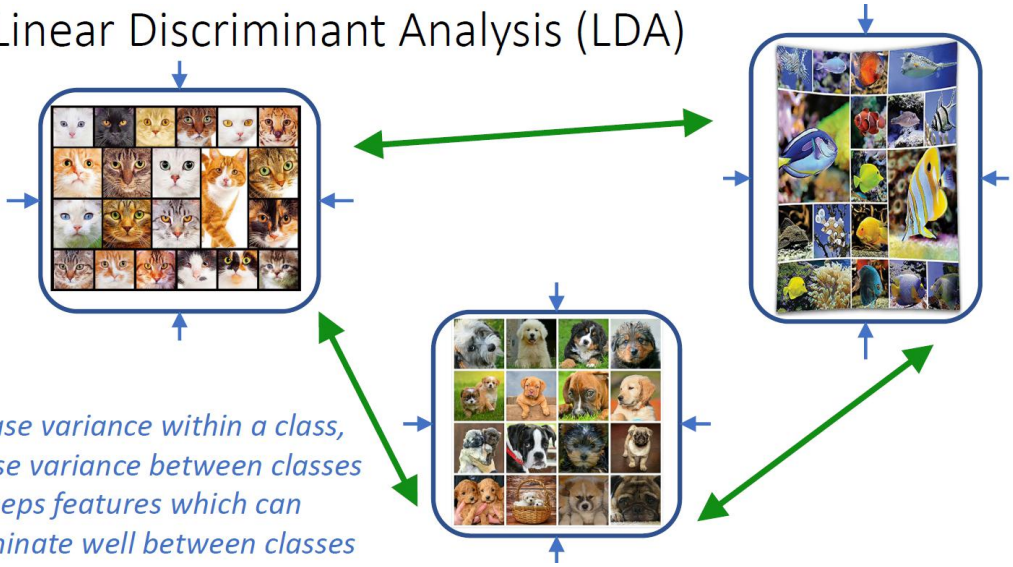
$$\underset{n\times f'}{\boldsymbol{X}_{\mathrm{PCA}}} = \underset{n\times f}{\boldsymbol{X}} \quad \underset{f\times f'}{\boldsymbol{U}[:,:f']}$$

$f' < f$

## LDA

need labels of all data, change data based on it's label

## Linear Discriminant Analysis (LDA)



*Decrease variance within a class, increase variance between classes*
*LDA keeps features which can discriminate well between classes*

## Linear Discriminant Analysis (LDA)

Intra (within) class scatter
Minimize

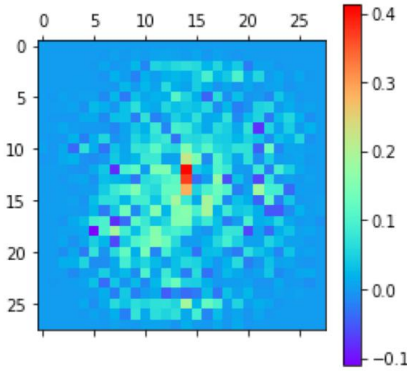
$$\boldsymbol{S}_{\text{intra}} = \sum_{c=1}^{C} \left(\tilde{\boldsymbol{X}}_c - \boldsymbol{\mu}_c\right)^T \left(\tilde{\boldsymbol{X}}_c - \boldsymbol{\mu}_c\right)$$

Inter (between) class scatter
Maximize

$$\boldsymbol{S}_{\text{inter}} = \sum_{c=1}^{C} N_c \left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)\left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)^T$$

$$\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V} = \text{SVD}\left\{\boldsymbol{S}_{\text{intra}}^{-1}\boldsymbol{S}_{\text{inter}}\right\}$$

$$\underset{n\times f'}{\boldsymbol{X}_{\text{LDA}}} = \underset{n\times f}{\boldsymbol{X}} \quad \underset{f\times f'}{\boldsymbol{U}[:,:f']}$$




C: Number of classes
$N_c$: Number of samples in class c
$\mu_c$: Feature mean of class c

# Hyper parameter

things Neural Network don't learning by themself

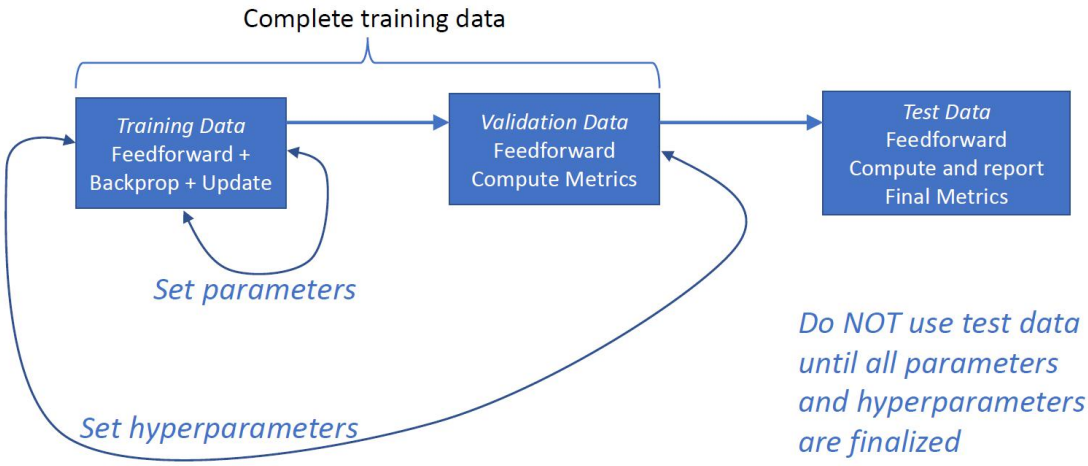
Continuous Examples:

- Learning rate η
- Momentum α
- Regularization λ

Discrete Examples:

- What optimizer: *SGD, Adam, RMSprop, …?*
- What regularization: *L2, L1, both, …?*
- What initialization: *Glorot, He, normal, uniform, …?*
- Batch size M: *32, 64, 128, …?*
- Number of MLP hidden layers: *1,2,3,…?*
- How many neurons in each layer?
- What kinds of data augmentation?

# Using data properly




*Do NOT use test data until all parameters and hyperparameters are finalized*

**use cross validation to set hyper-parameters**

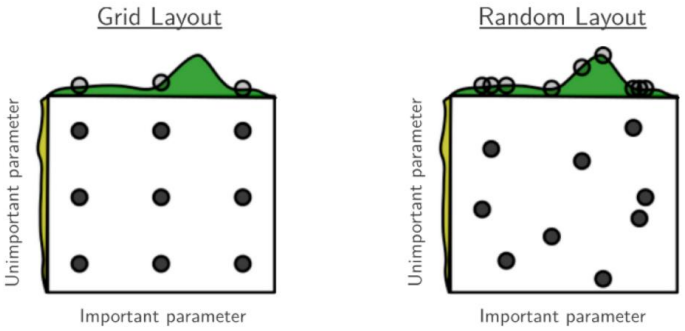
**Hyper-parameter seatch**

Grid search for (η,α):

(0.1,0.5), (0.1,0.9), (0.1,0.99),
(0.01,0.5), (0.01,0.9), (0.01,0.99),
(0.001,0.5), (0.001,0.9), (0.001,0.99)

Random search for (η,α):

(0.07,0.68), (0.002,0.94), (0.008,0.99),
(0.08,0.88), (0.005,0.62), (0.09,0.75),
(0.14,0.81), (0.006,0.76), (0.01,0.8)




*Random search samples more values of the important hyperparameter*

Pic courtesy: J. Bergstra, Y. Bengio, "Random Search for Hyperparameter

**Learning Rate Schedules**

$$\eta_{t+1} = \frac{\eta_0}{1 + kt}$$ ($\eta_0$ → Initial η; $t$ → Epochs)

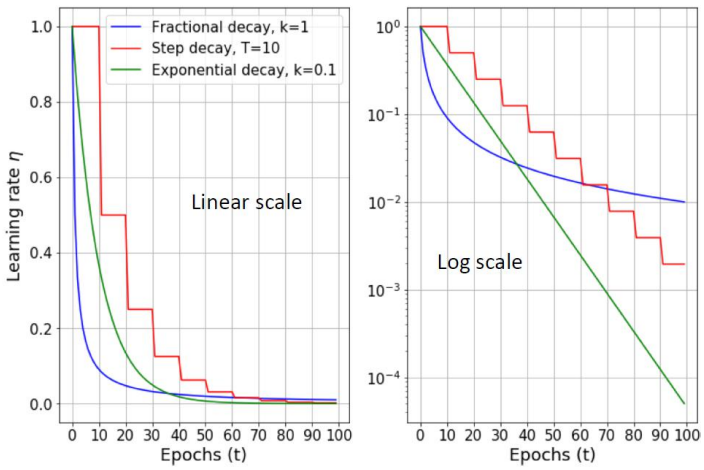
Fractional decay
Used in Keras

$$\eta_{T+1} = \eta_0 \left(\frac{1}{2}\right)^{\lfloor t/T \rfloor}$$

Step decay

$$\eta_{t+1} = \eta_0 e^{(-kt)}$$

Exponential decay

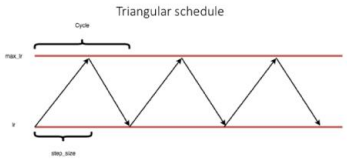


# Other Learning rate strategies

*Potential project topic*

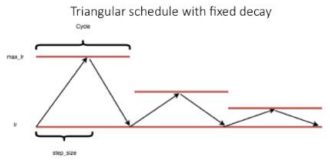
**Warmup:** Increase η for a few epochs at the beginning. Works well for large batch sizes.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". arXiv:1706.02677



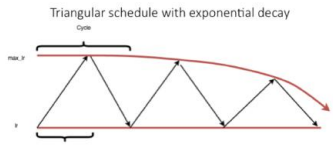

Cosine Scheduling

I. Loshchilov, F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", Proc. ICLR 2017.




Triangular Scheduling

L. N. Smith, "Cyclical Learning Rates for Training Neural Networks", arXiv:1506.01186
Pic courtesy https://www.jeremyjordan.me/nn-learning-rate/




# Universal Approximation Theorem

basic idea: Neural Network can simulate all function

http://neuralnetworksanddeeplearning.com/chap4.html

*Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let $I_{m_0}$ denote the $m_0$-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on $I_{m_0}$ is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there exist an integer $m_1$ and sets of real constants $\alpha_i$, $b_i$, and $w_{ij}$, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define*

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left( \sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \tag{4.88}$$

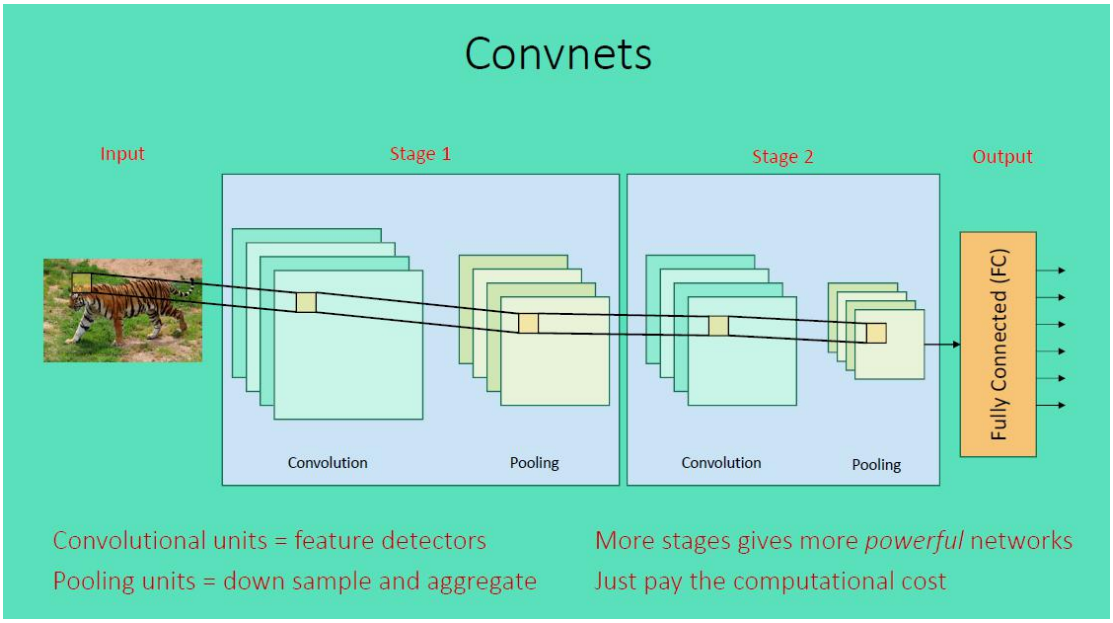
*as an approximate realization of the function $f(\cdot)$; that is,*

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \varepsilon$$

*for all $x_1, x_2, \ldots, x_{m_0}$ that lie in the input space.*

A single hidden layer MLP with squashing activation in the hidden layer and linear output layer can approximate any "engineering function"

# CNN




# Introduction

good introduction: http://cs231n.github.io/convolutional-networks/#pool

In summary:

A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)

There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)

Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function

Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)

Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

## Kernel(Filter)

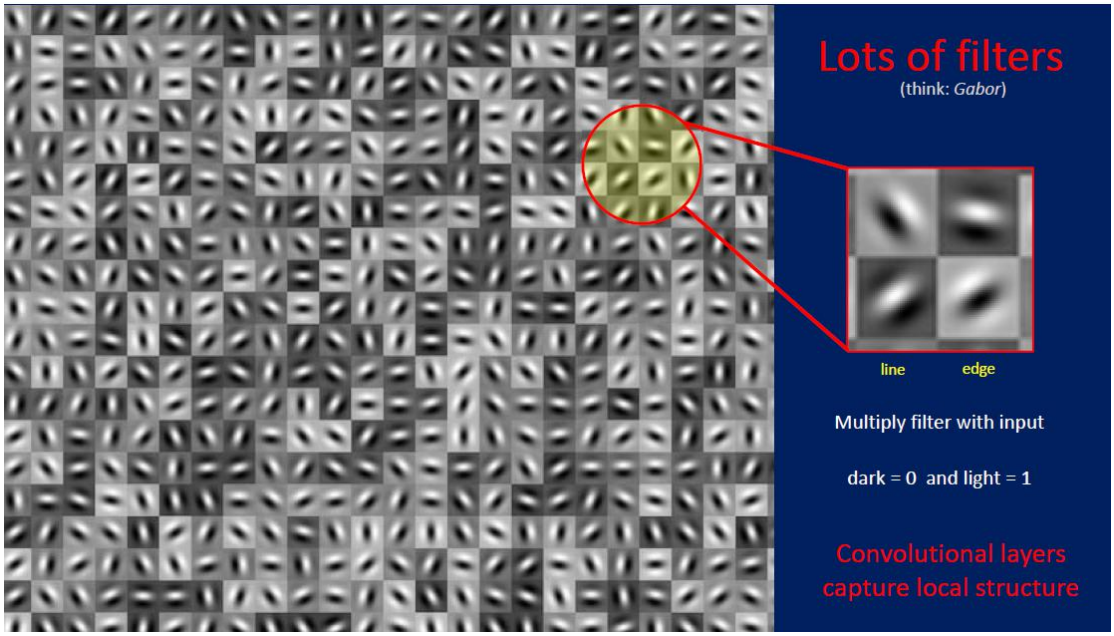
Slide

Padding

**size:**

small (3*3) deep

large (5*5) fast reduction,shallow, more parameters



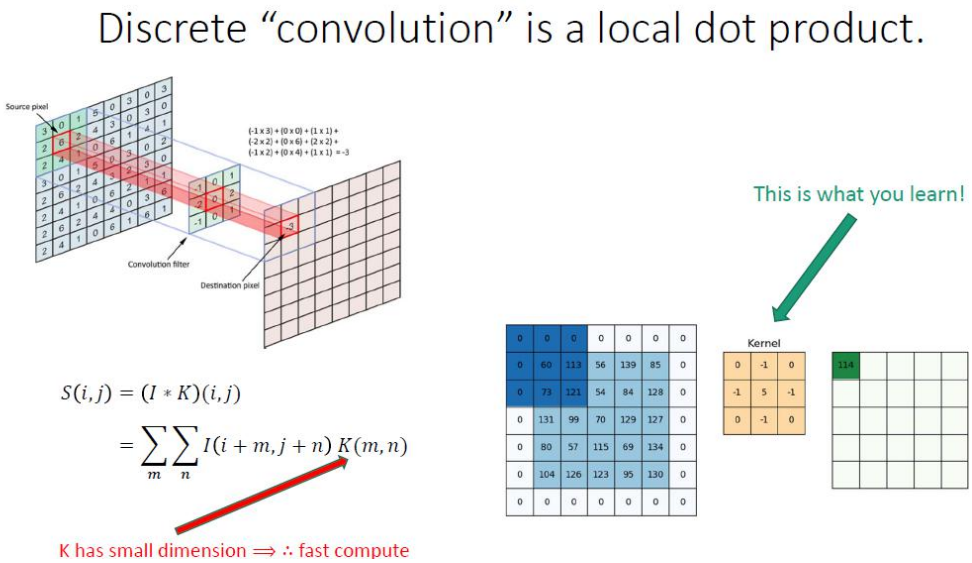

## Conv layer

image: 3-D in volume

**basic:** each neural is only connect to some of the neural in next layer

**Basic Function of Conv-layer:**

also like Σ(wx)+b (if **b** exist?)




use dot production to compute kernel* cut part

Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network.

**Hyper-Parameters**

**depth**

number of filters we want to use, they will look into the same region

**stride**

commonly--> 1

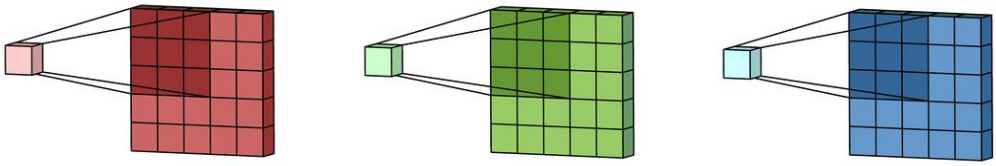
**zero-padding**

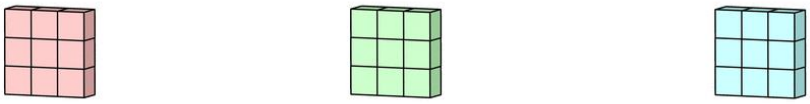
we could:

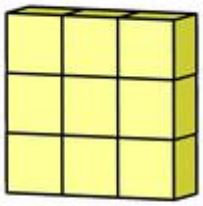
1. control the output volume size
2. remain the edge information

**For three channel**

three filters to combine a kernel, three output add up together, and add bias







**dilation**

it's possible to have filters that have spaces between each cell, called dilation.

in one dimension a filter *w* of size 3 would compute over input *x* the

following: *w[0]*x[0] + w[1]*x[1] + w[2]*x[2]*. This is dilation of 0. For dilation 1

the filter would instead compute *w[0]*x[0] + w[1]*x[2] + w[2]*x[4]*

so the computed block is not continous

**compute output size(O)**

by input size(**W**), filter size(**F**) and stride(**S**) and amount of zero padding(**P**)

$$O=(W-F+2P)/S+1.$$

input W*W output O*O

**Parameter Sharing(weight sharing)**

one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2). In other words, denoting a single 2-dimensional slice of depth as a **depth slice**

we are going to constrain the neurons in each depth slice to use the same weights and bias

for example: for a 11*11 kernel in RGB with 96 filters

there are 96*11*11*3+96 = 34944 parameters in totally

all 55*55 neurons in each depth slice will now be using the same parameters.

questions? each filter has three channel or just one?

matrix. If we were to use a kernel $\mathbf{K}$ of size 3 on the reshaped $4\times4$ input to get a $2\times2$ output, the equivalent transformation matrix would be:

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

**a new method: seperable convolution**

**Xception**

convert 3*3 kernel to 3*1 and 1*3 (used in Enmedded system)

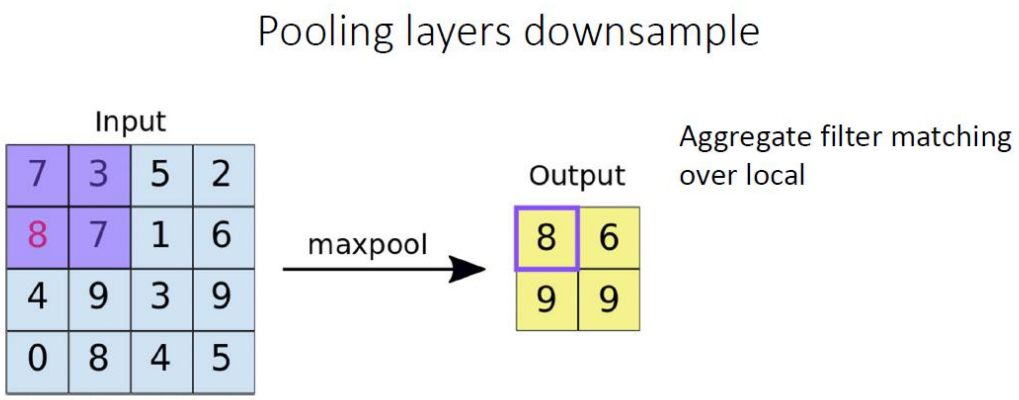
## Pooling layer

Its function is to progressively reduce the spatial size of the representation to **reduce the amount of parameters and computation in the network**, and hence to also control overfitting.

Max pooling | average pooling | L2-norm pooling

**Extents-->** # of points pooling together

**Stride -->** # of step to move in next pooling




Downsampling does more than just reduce size
It concentrates the image for deeper feature detection

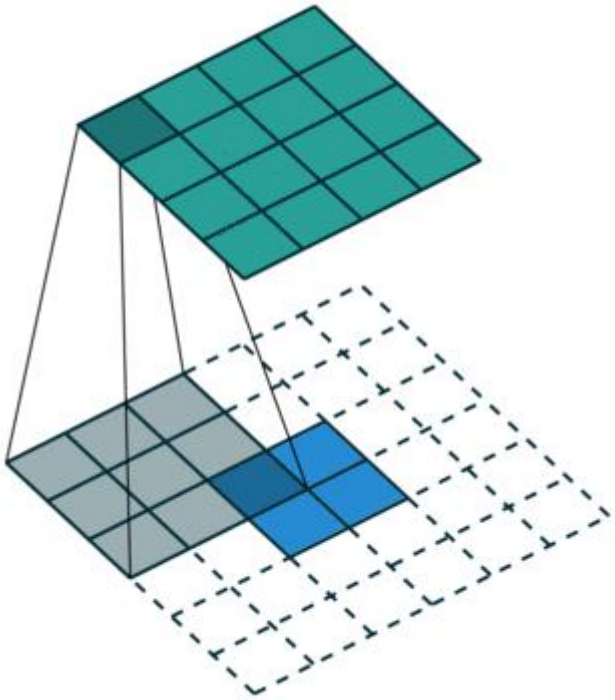
## De-Conv(Transpose convolution) layer

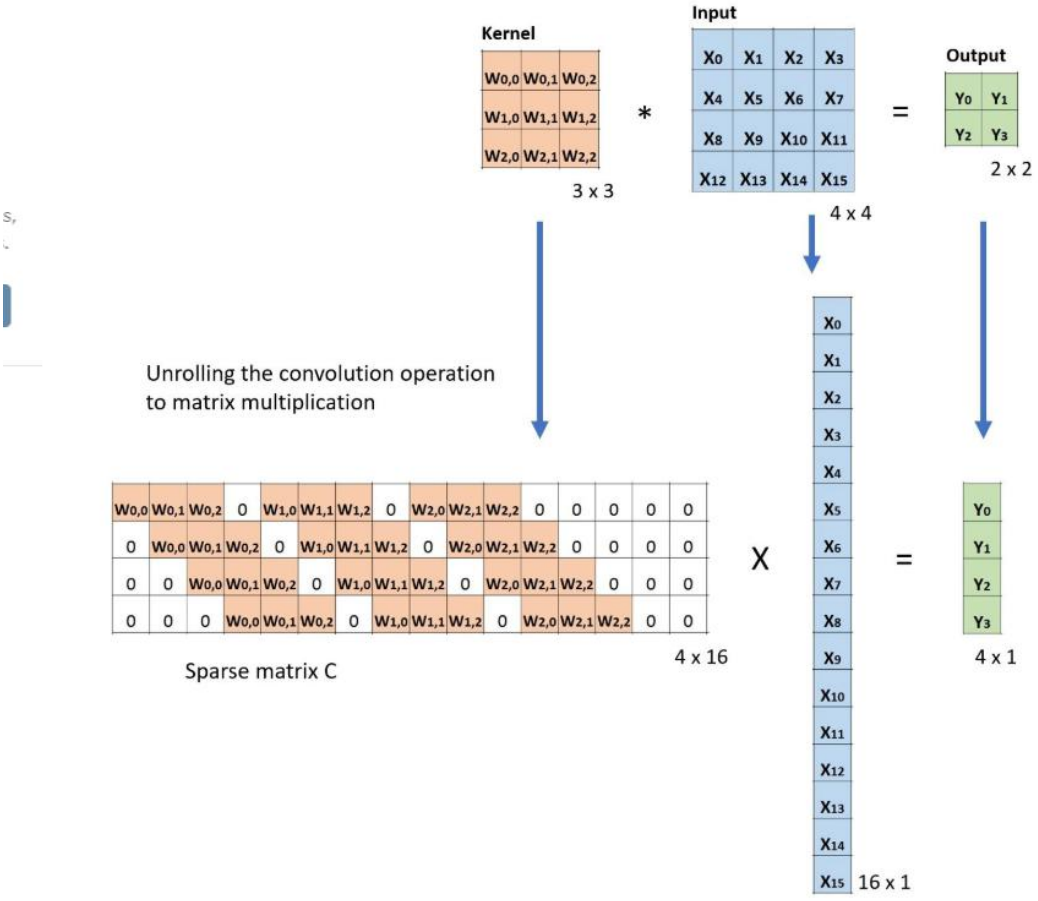
convolution: a pixels block into a point --> down sampling

de-convolution: a point to a block --> up sampling, from a low resolution to a higher one

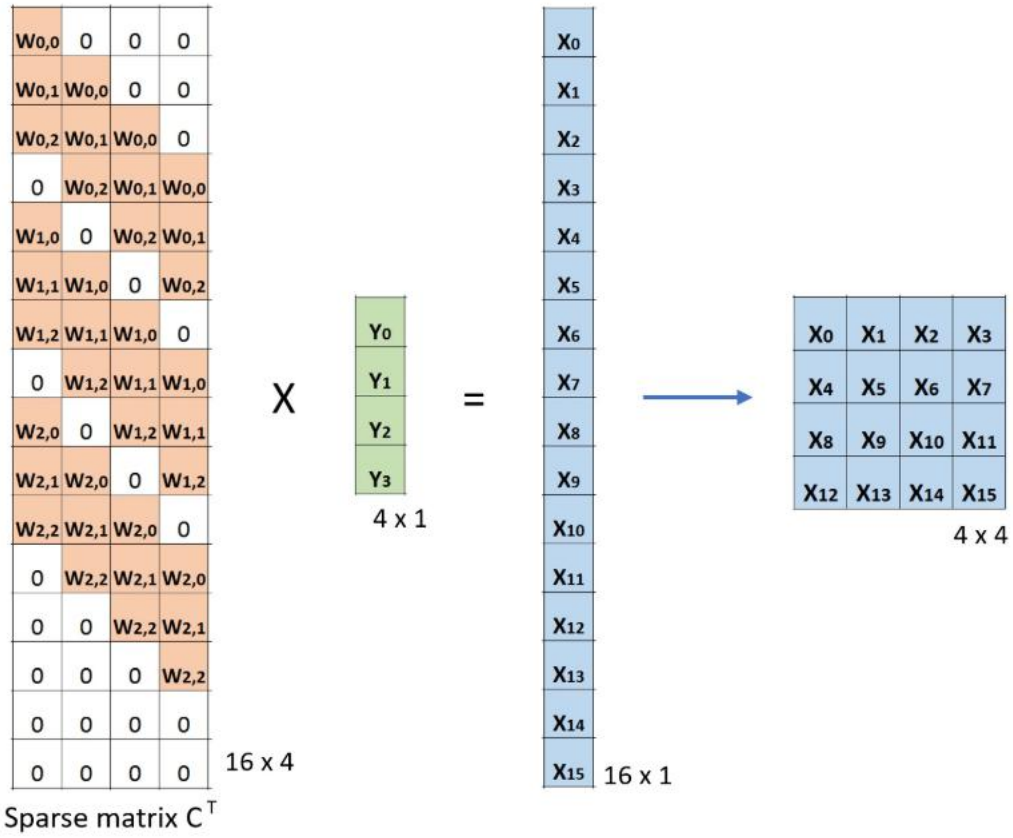
use one point in the previous layer to generate 3*3 blocks with the help of filter

For an example in the image below, we apply transposed convolution with a 3 x 3 kernel over a 2 x 2 input padded with a 2 x 2 border of zeros using unit strides. The up-sampled output is with size 4 x 4.



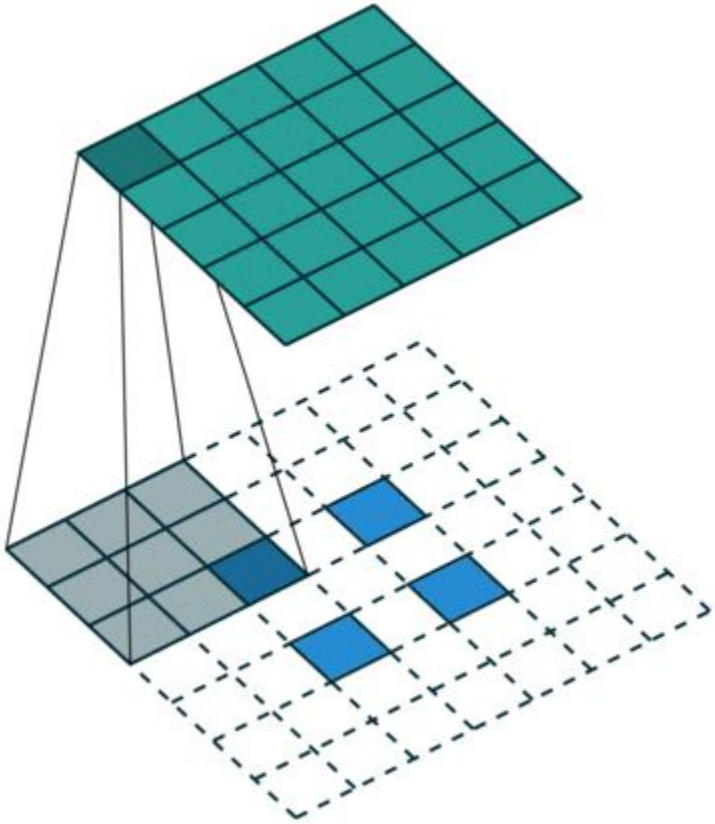



Matrix multiplication for convolution: from a Large input image (4 x 4) to a Small output image (2 x 2).




Matrix multiplication for convolution: from a Small input image (2 x 2) to a Large output image (4 x 4).

map e.g. 4-dimensional space to 25-dimensional space



Transposed 2D convolution with no padding, stride of 2 and kernel of 3

## Group Convolution

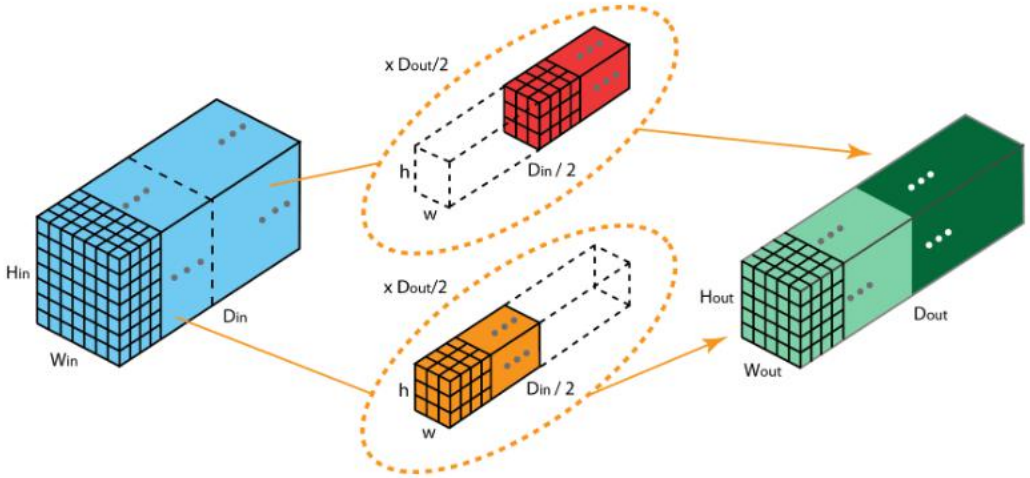
In each filter group, the depth of each filter is only half of the that in the nominal 2D convolutions. They are of depth Din / 2. Each filter group contains Dout /2 filters. The first filter group (red) convolves with the first half of the input layer ([:, :, 0:Din/2]), while the second filter group (blue) convolves with the second half of the input layer ([:, :, Din/2:Din]). As a result, each filter group creates Dout/2 channels. Overall, two groups create 2 x Dout/2 = Dout channels. We then stack these channels in the output layer with Dout channels.

**Advantage:**

efficient training.

model is more efficient

Grouped convolution may provide a better model than a nominal 2D convolution.




Grouped convolution with 2 filter groups.

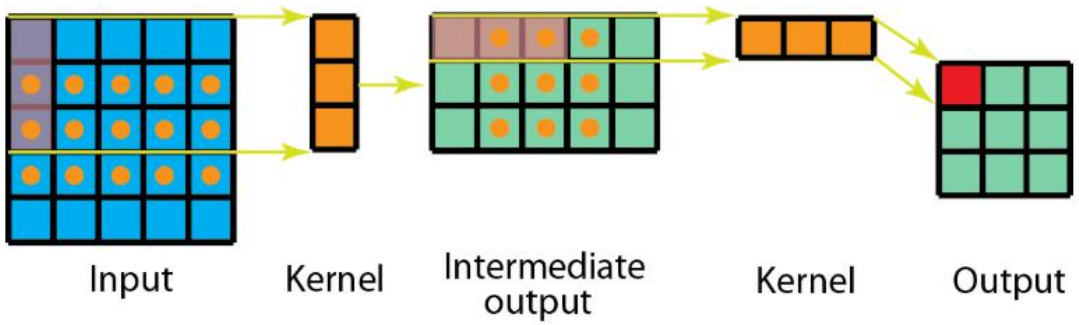
**this is a bit similar like seperable convolutions**

If the number of filter groups is the same as the input layer channel, each filter is of depth Din / Din = 1. This is the same filter depth as in depthwise convolution.

## Seperable Convolutions

# Efficiency!

in **MobileNet:** https://arxiv.org/pdf/1704.04861.pdf
change the 3*3 kernel to 3*1 and 1*3 two kernel This would require 6 instead of 9 parameters while doing the same operations.




Spatially separable convolution with 1 channel.

Although spatially separable convolutions save cost, it is rarely used in deep learning. One of the main reason is that not all kernels can be divided into two, smaller kernels. If we replace all traditional convolutions by the spatially separable convolution, we limit ourselves for searching all possible kernels during training. The training results may be sub-optimal.

**Depthwise Separable Convolutions**

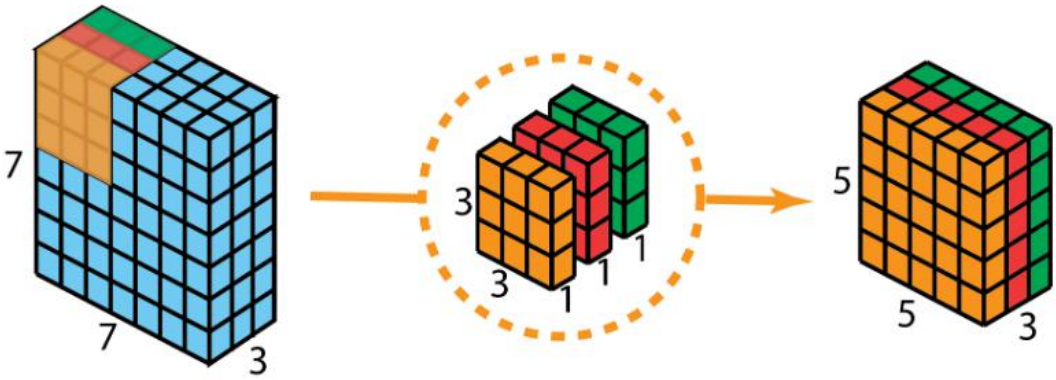
much commonly used in deep learning: **depthwise convolution** and **1*1 convolution**

First, we apply **depthwise convolution** to the input layer. Instead of using a single filter of size 3 x 3 x 3 in 2D convolution, we used **3 kernels**, **separately.** Each filter has size **3 x 3 x 1**. Each kernel convolves with 1 channel of the input layer (1 channel only, not all channels!). We then

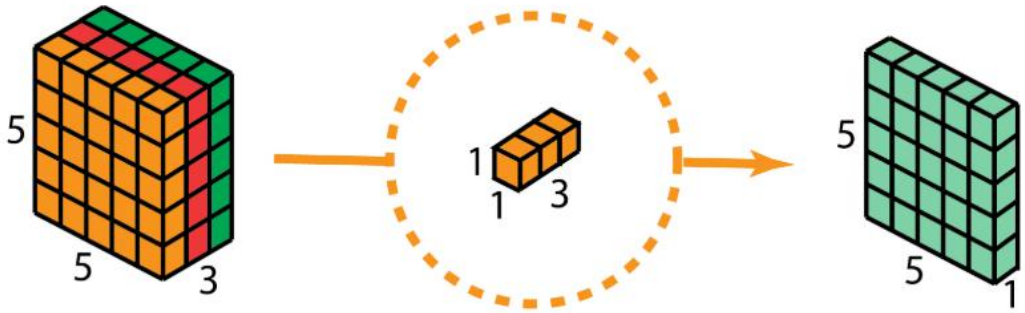
stack these maps together to create a 5 x 5 x 3 image. After this, we have the output with size 5 x 5 x 3.

As the second step of depthwise separable convolution, to extend the depth, we apply the 1x1 convolution with kernel size 1x1x3. Convolving the 5 x 5 x 3 input image with each 1 x 1 x 3 kernel provides a map of size 5 x 5 x 1.

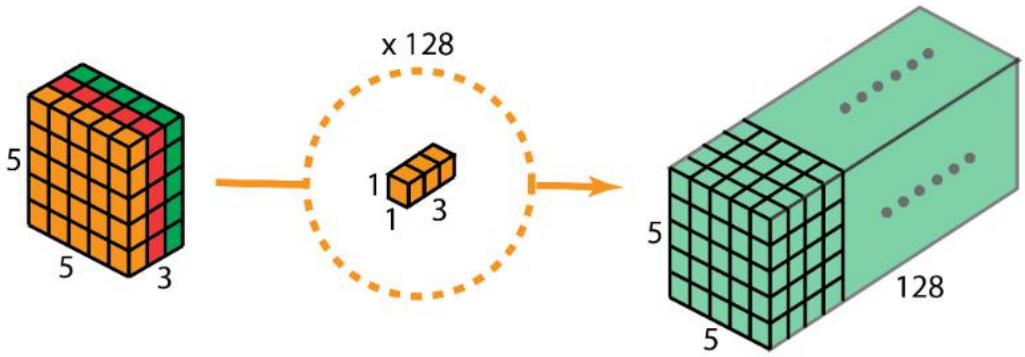
Thus, after applying 128 1x1 convolutions, we can have a layer with size 5 x 5 x 128.




Depthwise separable convolution — first step: Instead of using a single filter of size 3 x 3 x 3 in 2D convolution, we used 3 kernels, separately. Each filter has size 3 x 3 x 1. Each kernel convolves with 1 channel of the input layer (1 channel only, not all channels!). Each of such convolution provides a map of size 5 x 5 x 1. We then stack these maps together to create a 5 x 5 x 3 image. After this, we have the output with size 5 x 5 x 3.




Thus, after applying 128 1x1 convolutions, we can have a layer with size 5 x 5 x 128.




Depthwise separable convolution — second step: apply multiple 1 x 1 convolutions to modify depth.

**drwaback:**

reduces the number of parameters in the convolution. As such, for a small model, the model capacity may be decreased significantly if the 2D convolutions are replaced by depthwise separable convolutions.

### Full Connect layer

## # of Parameters computation

## Visualization

layers that are deeper in the network visualize more training data specific features, while the earlier layers tend to visualize general patterns like edges, texture, background

**Visualize different layer**

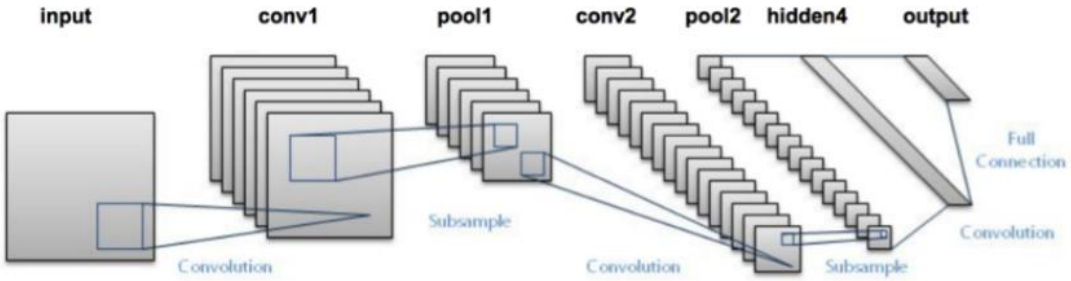
**Visualize filter**

**Heatmap**

## Different Nets

### LeNet(1998)

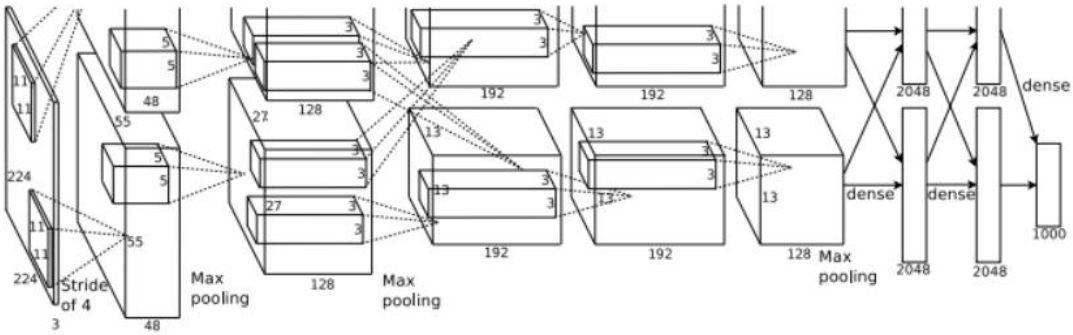
classifies digits in 32x32 pixel greyscale inputimages




## AlexNet(2012)

deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum.

train on two GPUs

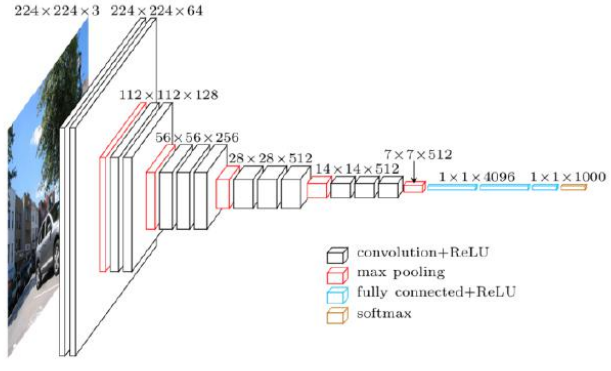



## VGG

https://arxiv.org/pdf/1409.1556.pdf

Baseline feature extractor

16 (weighted) layers CNN  138M parameters




| Name: | A | A-LRN | B | C | D | E |
|---|---|---|---|---|---|---|
| # Layers | 11 | 11 | 13 | 16 | 16 | 19 |
| | C3D64 | C3D64 | C3D64 | C3D64 | C3D64 | C3D64 |
| | | LRN | C3D64 | C3D64 | C3D64 | C3D64 |
| | M | M | M | M | M | M |
| | C3D128 | C3D128 | C3D128 | C3D128 | C3D128 | C3D128 |
| | | | C3D128 | C3D128 | C3D128 | C3D128 |
| | M | M | M | M | M | M |
| | C3D256 | C3D256 | C3D256 | C3D256 | C3D256 | C3D256 |
| | C3D256 | C3D256 | C3D256 | C3D256 | C3D256 | C3D256 |
| | | | | C1D256 | C3D256 | C3D256 |
| | | | | | | C3D256 |
| | M | M | M | M | M | M |
| | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 |
| | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 |
| | | | | C1D512 | C3D512 | C3D512 |
| | | | | | | C3D512 |
| | M | M | M | M | M | M |
| | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 |
| | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 | C3D512 |
| | | | | C1D512 | C3D512 | C3D512 |
| | | | | | | C3D512 |
| | M | M | M | M | M | M |
| | FC4096 | FC4096 | FC4096 | FC4096 | FC4096 | FC4096 |
| | FC4096 | FC4096 | FC4096 | FC4096 | FC4096 | FC4096 |
| | FC1000 | FC1000 | FC1000 | FC1000 | FC1000 | FC1000 |
| | S | S | S | S | S | S |

Original training: 3 weeks on 4 GPUs

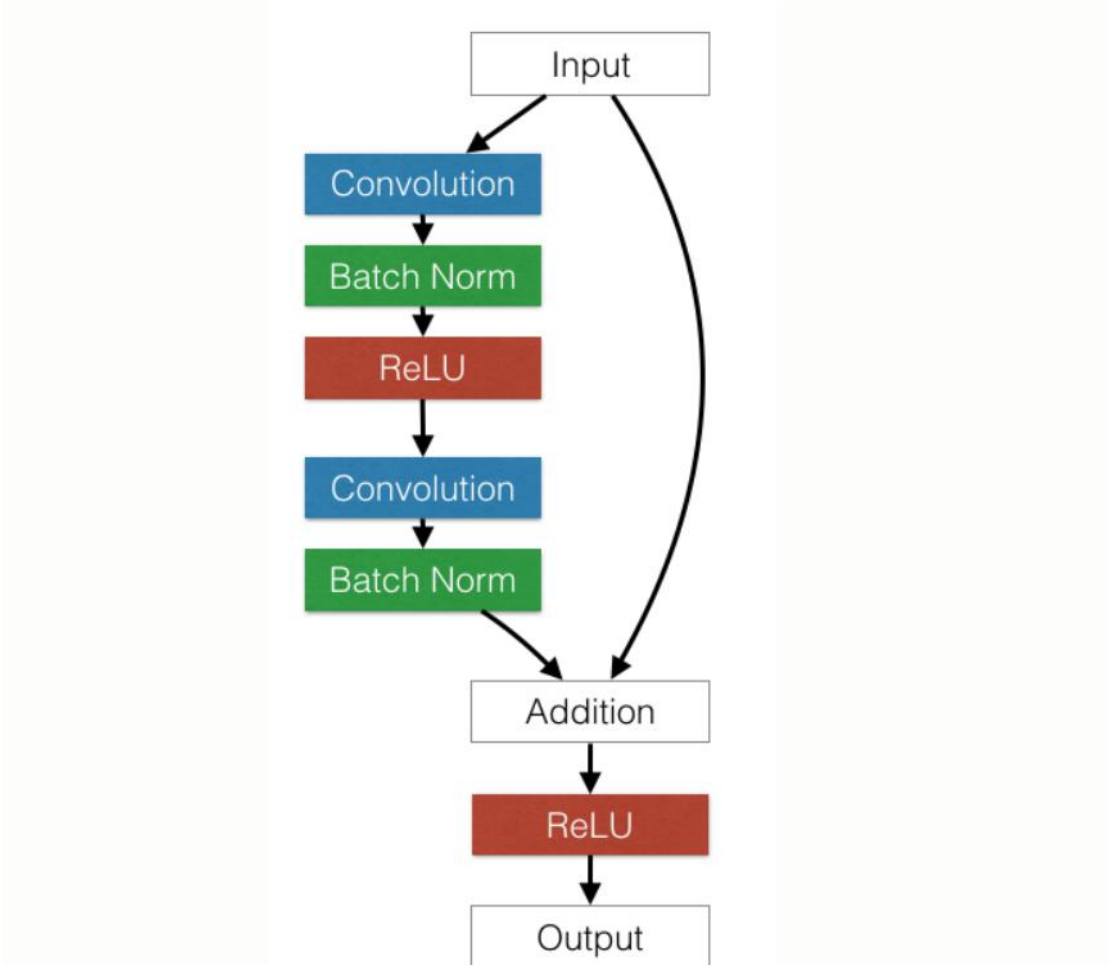
*# parameters ≈ 138M*

## Resnet

gated recurrent units

take a standard feed-forward ConvNet and add **skip connections** that bypass (or shortcut) a few convolution layers at a time. Each bypass gives rise to a residual block in which the convolution layers predict a residual that is added to the block's input tensor.

aim to avoid vanish gradient




## GoogleNet (Inception)

https://arxiv.org/pdf/1512.00567.pdf
https://arxiv.org/pdf/1409.4842.pdf
no Pooling layer

**Introduction**

The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. It used batch normalization, image distortions and RMSprop. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million.

**Speciality**

1. use several small filter to stand large filter --> n*1 + 1*n to replace n*n
2. for a single input layer, applied many different filters(some are pooling, some are 1*1 with 1*n with n*1) and concatedate result (add together)

---> **to avoid representational bottlenecks and avoid stop locally**

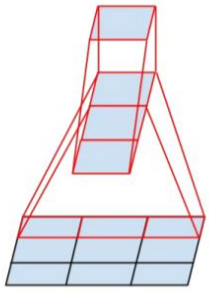


Figure 3. Mini-network replacing the $3 \times 3$ convolutions. The lower layer of this network consists of a $3 \times 1$ convolution with 3 output units.

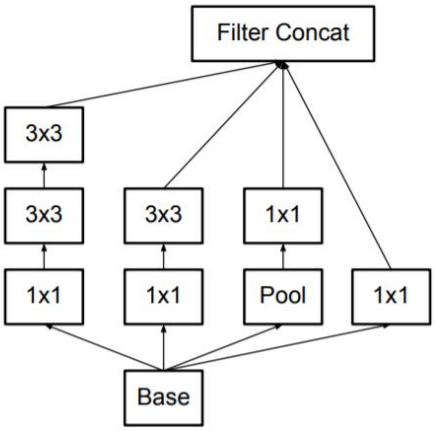
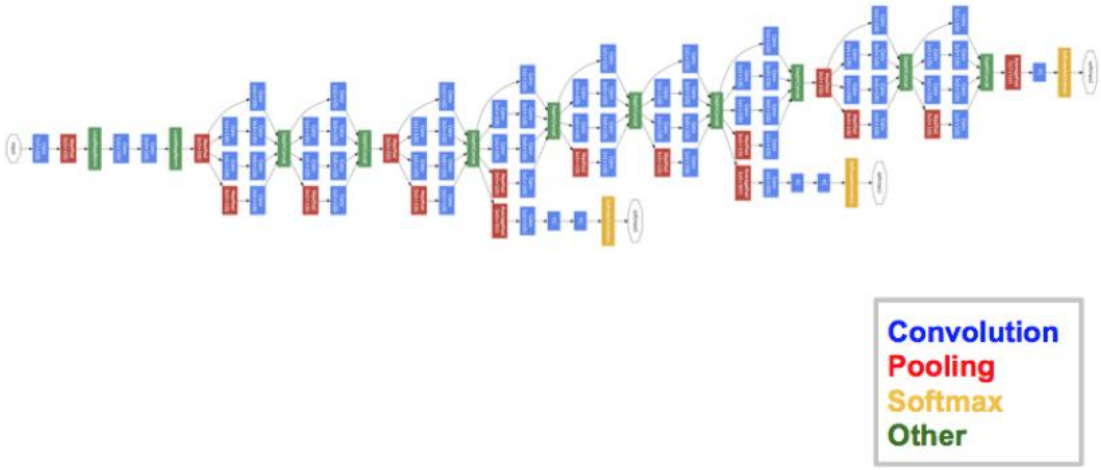



Figure 5. Inception modules where each $5 \times 5$ convolution is replaced by two $3 \times 3$ convolution, as suggested by principle 3 of Section 2.




## MobileNet

https://arxiv.org/pdf/1704.04861.pdf

Deepwise Seperable Convolution + Pointwise Convolution (1*1)

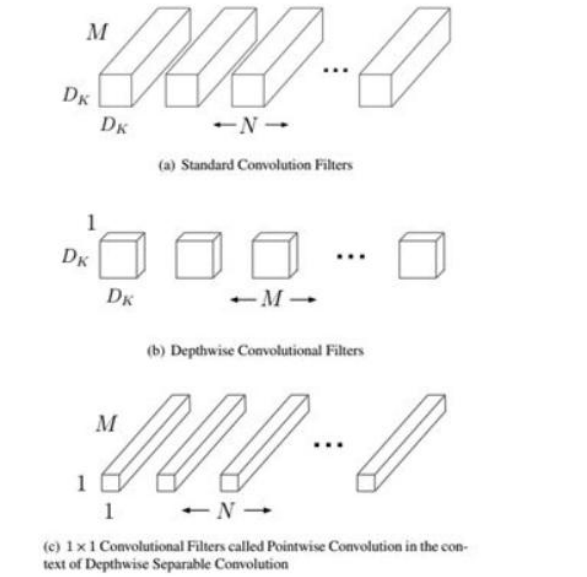



图1 Depthwise separable convolution

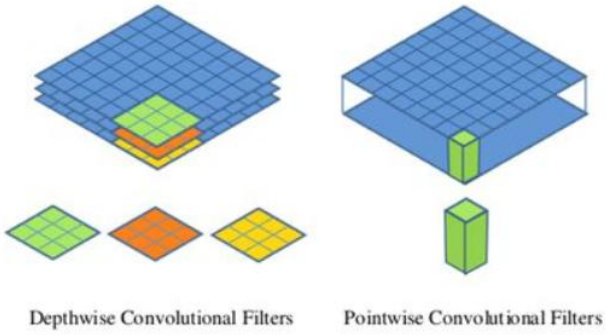



图2 Depthwise convolution和pointwise convolution

## MobileNet网络结构

前面讲述了depthwise separable convolution，这是MobileNet的基本组件，但是在真正应用中会加入batchnorm，并使用ReLU激活函数，所以depthwise separable convolution的基本结构如图3所示。

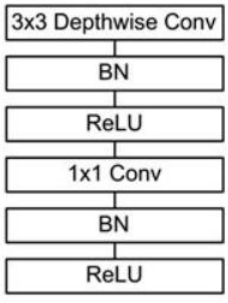



图3 加入BN和ReLU的depthwise separable convolution

MobileNet 的网络结构如表 1 所示。首先是一个 3x3 的标准卷积，然后后面就是堆积 depthwise separable convolution，并且可以看到其中的部分 depthwise convolution 会通过 strides=2 进行 down sampling。然后采用 average pooling 将 feature 变成 1x1，根据预测类别大小加上全连接层，最后是一个 softmax 层。如果单独计算 depthwise convolution 和 pointwise convolution，整个网络有 28 层（这里 Avg Pool 和 Softmax 不计算在内）。我们还可以分析整个网络的参数和计算量分布，如表 2 所示。可以看到整个计算量基本集中在 1x1 卷积上，如果你熟悉卷积底层实现的话，你应该知道卷积一般通过一种 im2col 方式实现，其需要内存重组，但是当卷积核为 1x1 时，其实就不需要这种操作了，底层可以有更快的实现。对于参数也主要集中在 1x1 卷积，除此之外还有就是全连接层占了一部分参数。

### MobileNet网络结构

前面讲述了depthwise separable convolution，这是MobileNet的基本组件，但是在真正应用中会加入batchnorm，并使用ReLU激活函数，所以depthwise separable convolution的基本结构如图3所示。

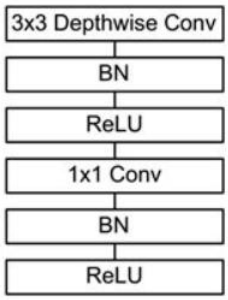



图3 加入BN和ReLU的depthwise separable convolution

| Type / Stride | Filter Shape | Input Size |
|---|---|---|
| Conv / s2 | 3 × 3 × 3 × 32 | 224 × 224 × 3 |
| Conv dw / s1 | 3 × 3 × 32 dw | 112 × 112 × 32 |
| Conv / s1 | 1 × 1 × 32 × 64 | 112 × 112 × 32 |
| Conv dw / s2 | 3 × 3 × 64 dw | 112 × 112 × 64 |
| Conv / s1 | 1 × 1 × 64 × 128 | 56 × 56 × 64 |
| Conv dw / s1 | 3 × 3 × 128 dw | 56 × 56 × 128 |
| Conv / s1 | 1 × 1 × 128 × 128 | 56 × 56 × 128 |
| Conv dw / s2 | 3 × 3 × 128 dw | 56 × 56 × 128 |
| Conv / s1 | 1 × 1 × 128 × 256 | 28 × 28 × 128 |
| Conv dw / s1 | 3 × 3 × 256 dw | 28 × 28 × 256 |
| Conv / s1 | 1 × 1 × 256 × 256 | 28 × 28 × 256 |
| Conv dw / s2 | 3 × 3 × 256 dw | 28 × 28 × 256 |
| Conv / s1 | 1 × 1 × 256 × 512 | 14 × 14 × 256 |
| 5× Conv dw / s1 | 3 × 3 × 512 dw | 14 × 14 × 512 |
| 5× Conv / s1 | 1 × 1 × 512 × 512 | 14 × 14 × 512 |
| Conv dw / s2 | 3 × 3 × 512 dw | 14 × 14 × 512 |
| Conv / s1 | 1 × 1 × 512 × 1024 | 7 × 7 × 512 |
| Conv dw / s2 | 3 × 3 × 1024 dw | 7 × 7 × 1024 |
| Conv / s1 | 1 × 1 × 1024 × 1024 | 7 × 7 × 1024 |
| Avg Pool / s1 | Pool 7 × 7 | 7 × 7 × 1024 |
| FC / s1 | 1024 × 1000 | 1 × 1 × 1024 |
| Softmax / s1 | Classifier | 1 × 1 × 1000 |

表1 MobileNet的网络结构

## Comparation

models in the context of image classification. We measured the time for a full forward and backward pass for a mini-batch of 32 images for ResNet, VGG A, VGG D, Batch-Normalized Inception, and Inception v3 on a NVIDIA Titan X. Also listed is the top-1 single-crop validation error on the Imagenet-2012 dataset.

| Model | Top-1 err (%) | Time (ms) |
|---|---|---|
| VGG-A | 29.6 | 372 |
| VGG-D | 26.8 | 687 |
| **ResNet-34** | 26.7 | 231 |
| BN-Inception | 25.2 | 192 |
| **ResNet-50** | 24.0 | 403 |
| **ResNet-101** | 22.4 | 649 |
| Inception-v3 | 21.2 | 494 |

# CNN in NLP

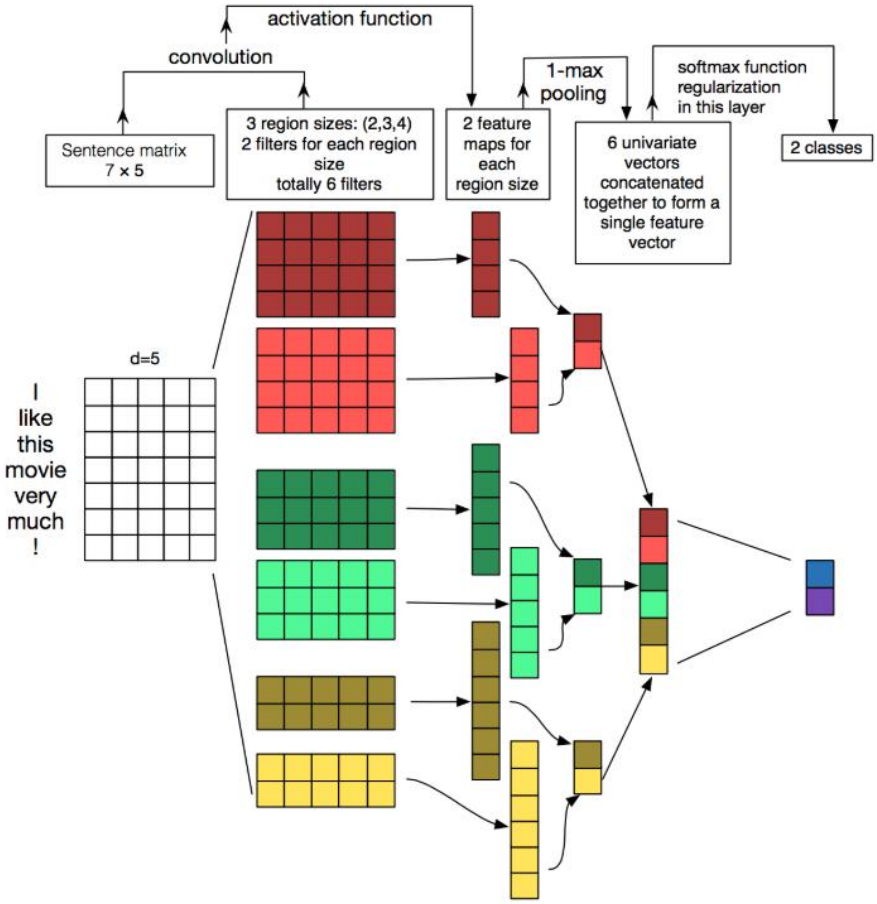



Image Reference : http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

# Coding

## Keras

a basic model

from keras.models import Sequential  from keras.layers import Dense, Conv2D, Flatten#create model  model = Sequential()#add model layers  model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(28,28,1)))  model.add(Conv2D(32, kernel_size=3, activation='relu'))  model.add(Flatten())  model.add(Dense(10, activation='softmax'))  # Compiling the model takes three parameters: optimizer, loss and metrics. model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])  #train the model  model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3)  #predict first 4 images in the test set  model.predict(X_test[:4])  model = Sequential() model.add(Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]))  model.add(Activation('relu'))  model.add(Conv2D(32, (3, 3))) model.add(Activation('relu'))  model.add(MaxPooling2D(pool_size=(2, 2))) model.add(Dropout(0.25))  model.add(Conv2D(64, (3, 3), padding='same')) model.add(Activation('relu'))  model.add(Conv2D(64, (3, 3)))  model.add(Activation('relu')) model.add(MaxPooling2D(pool_size=(2, 2)))  model.add(Dropout(0.25))  model.add(Flatten())

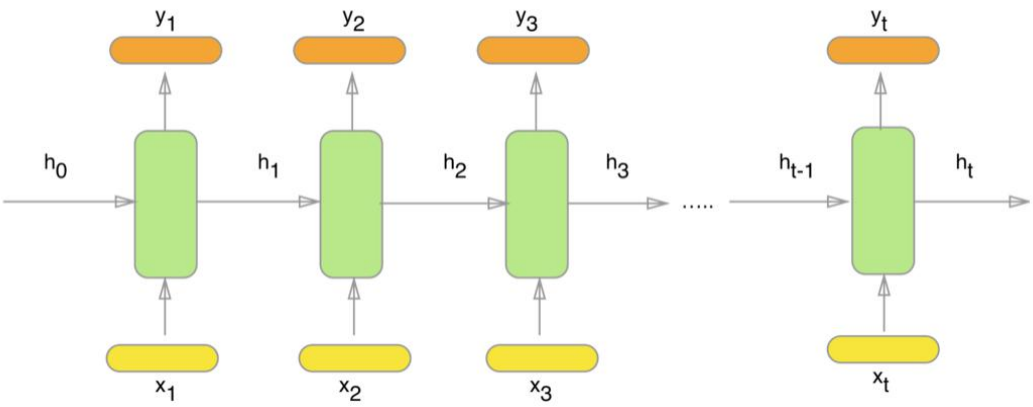
model.add(Dense(512)) model.add(Activation('relu')) model.add(Dropout(0.5))
model.add(Dense(num_classes)) model.add(Activation('softmax'))

**'Flatten' layer.** Flatten serves as a connection between the convolution and dense layers.

# RNN

## Introduction

*x_1, x_2, x_3, ..., x_t* represent the input words from the text, *y_1, y_2, y_3, ..., y_t* represent the predicted next words and *h_0, h_1, h_2, h_3, ..., h_t* hold the information for the previous input words.




1) $h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
2) $y_t = softmax(W^{(S)}h_t)$
3) $J^{(t)}(\theta) = \sum_{i=1}^{|V|} (y_{t_i}^{'} log y_{t_i})$

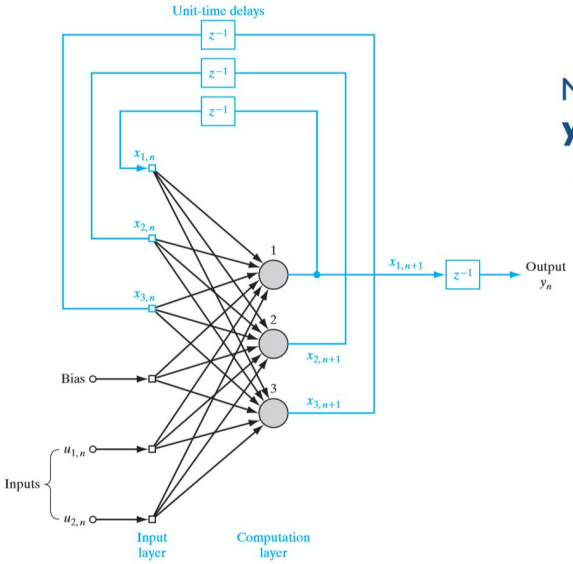
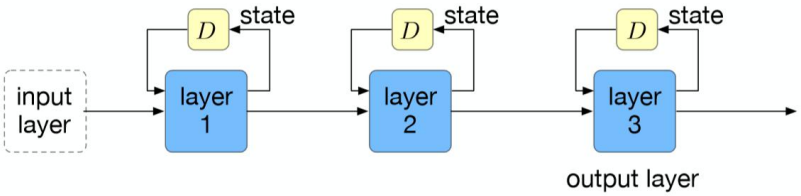



FIGURE 15.6 Fully connected recurrent network with two inputs, two hidden neurons, and one output neuron. The feedback connections are shown in red to emphasize their global role.

Network has **state** — current output **y**[n] is a function of current input **x**[n] and state (e.g., previous **x**[n-i], **y**[n-i]

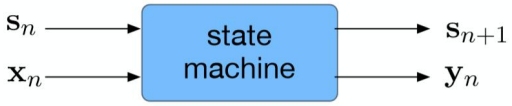
*think of this as a generalization of a feedback filter*

basic structure




**STATE**

state machine: network is stated:




$$\mathbf{s}_{n+1} = \text{next\_state}(\mathbf{s}_n, \mathbf{x}_n)$$
$$\mathbf{y}_n = \text{output}(\mathbf{s}_n, \mathbf{x}_n)$$

*these two functions define the state machine*

$\mathbf{s}_n$ state at time n (before input x_n is applied)

$\mathbf{x}_n$ input at time n

$\mathbf{s}_{n+1}$ state at time n+1 (after input x_n is applied)

$\mathbf{y}_n$ output at time n

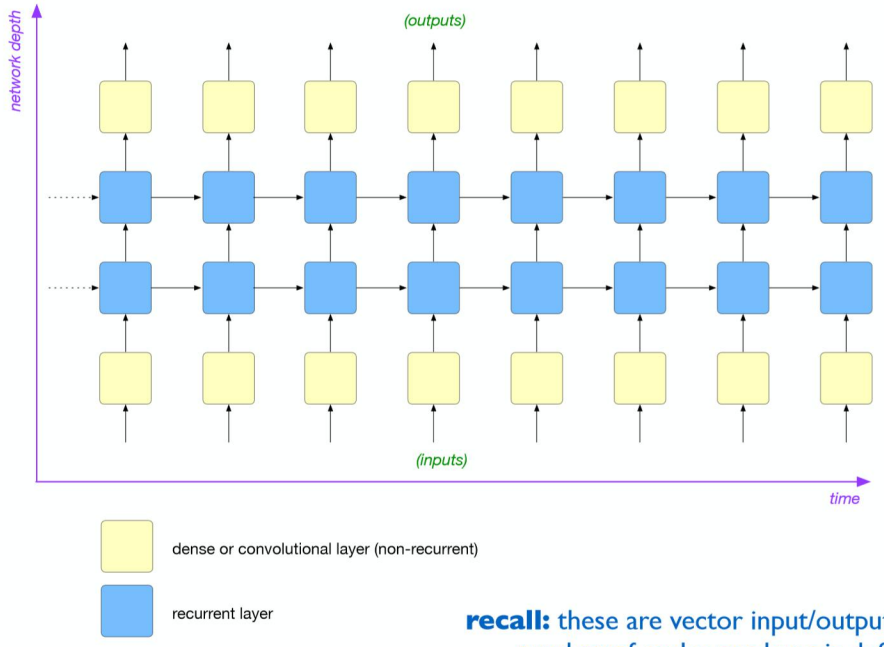
(Vanilla RNN)

Keras code

keras.layers.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None,

bias_constraint=None, dropout=0.0, recurrent_dropout=0.0, return_sequences=False, return_state=False, go_backwards=False, stateful=False, unroll=False)

**Diagram of Neural Net**




**Number of nodes and parameters in each layer (blue block on upper diagram)**

$$\mathbf{s}_{n+1} = \underline{h}(\mathbf{V}\mathbf{s}_n + \mathbf{W}\mathbf{x}_n + \mathbf{b})$$
$$\mathbf{y}_n = \mathbf{s}_{n+1}$$

`keras.layers.SimpleRNN(units)`

$$\underset{N\times 1}{\mathbf{y}_n} \quad \underset{N\times M}{\mathbf{W}} \quad \underset{M\times 1}{\mathbf{x}_n}$$

$$\underset{N\times 1}{\mathbf{s}_{n+1}} \quad \underset{N\times N}{\mathbf{V}} \quad \underset{N\times 1}{\mathbf{s}_n}$$

N: number of "units" (nodes) in this recurrent layer

M: number of "units" (nodes) in the previous layer

Node on layer: n

Node on previous layer: m

input: m

output: n

**parameters**: **(m+n)*n** (weight) **+n** (bias)

When training, a training window length T is selected =>

a sequence of input of length T

a sequence of label of length T

http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

# Vanish Gradient Problem

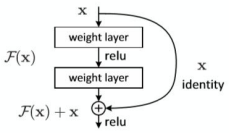
Vanishing gradient is a training-based point of view of a more fundamental issue.

The real issue is **decaying influence** or **degradation**

*basically, inputs to repeated linear-nonlinear operations cannot affect outputs far into the "future"*

**Feedfoward networks**:"future" means deeper in the network

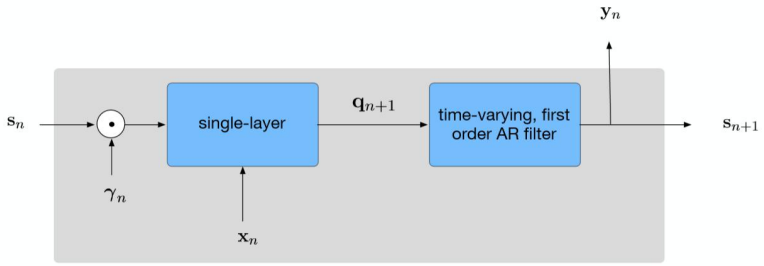
"degradation" coined by ResNet authors and Residual connections introduced to go deeper




**GATE**

To solve this problem we need to add GATE which is attenuating and/or filtering in the state update equation (my understanding: amplify the influence of previous state)

## Generic RNN with Gates



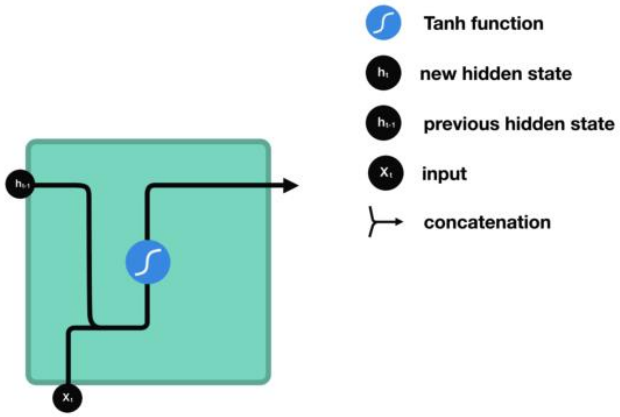

$$\mathbf{q}_{n+1} = \underline{h}(\mathbf{V}[\boldsymbol{\gamma}_n \odot \mathbf{s}_n] + \mathbf{W}\mathbf{x}_n + \mathbf{b}) \qquad \mathbf{s}_{n+1} = \boldsymbol{\alpha}_n \odot \mathbf{s}_n + \boldsymbol{\beta}_n \odot \mathbf{q}_{n+1}$$

$\boldsymbol{\gamma}_n$ "read gate" — attenuation on current state when read for update

$\boldsymbol{\alpha}_n$ "forget gate" — feedback coefficients for state filtering

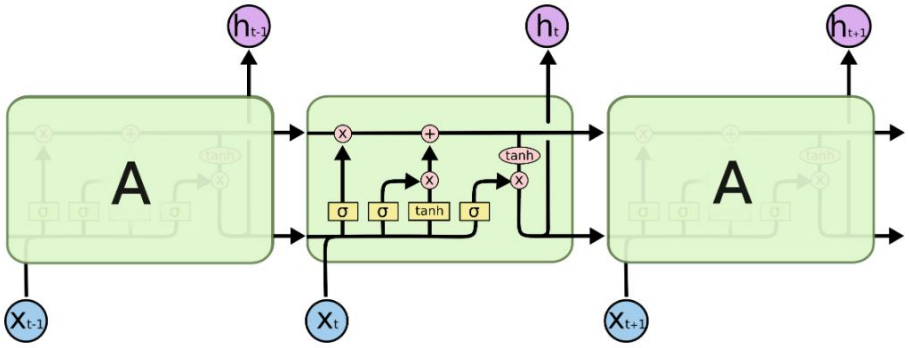
$\boldsymbol{\beta}_n$ "write gate" — feedforward coefficients for state filtering

All gate are trainable parameters and are learned using a single layer feedforward network (my understanding: GATE make simple m*n parameters network more complex, though if we block the process, it is still a m input and n output problem, the inner parametes(weight) are no longer n(m+n+1), it because more complex, but still based on **V(n*n), W(m*n), b (n)**

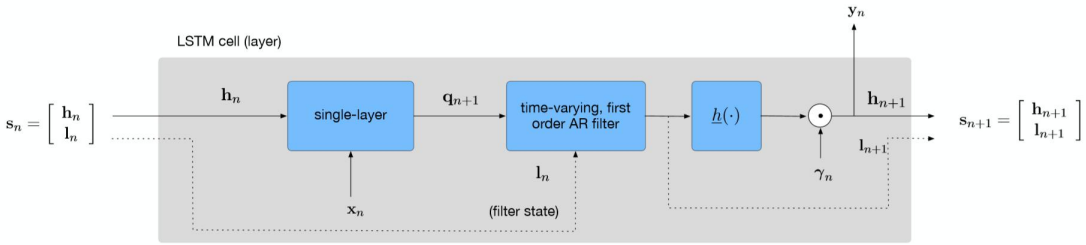



tanh make the output in between -1 -> 1

## LSTM




# Long/Short-Term Memory (LSTM) Cell




$$\mathbf{q}_{n+1} = \underline{h}(\mathbf{V}\mathbf{h}_n + \mathbf{W}\mathbf{x}_n + \mathbf{b}) \qquad \mathbf{h}_{n+1} = \boldsymbol{\gamma}_n \odot \underline{h}(\mathbf{l}_{n+1})$$

$$\mathbf{l}_{n+1} = \boldsymbol{\alpha}_n \odot \mathbf{l}_n + \boldsymbol{\beta}_n \odot \mathbf{q}_{n+1}$$

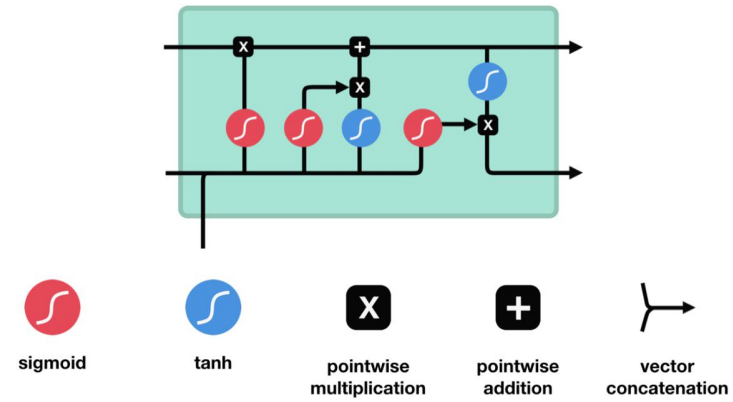
$$\mathbf{s}_n = \begin{bmatrix} \mathbf{h}_n \\ \mathbf{l}_n \end{bmatrix} = \begin{bmatrix} \text{short-term state} \\ \text{long-term state} \end{bmatrix}$$

Differences from GRU:

"forget gate" and "write gate" decoupled ~ different alpha and beta

"read gate" applied on output (gates read of next cell at n+1)

activation applied on the long-term state before applying read gate




LSTM Cell and It's Operations

These operations are used to allow the LSTM to keep or forget information.

**Core Concept**

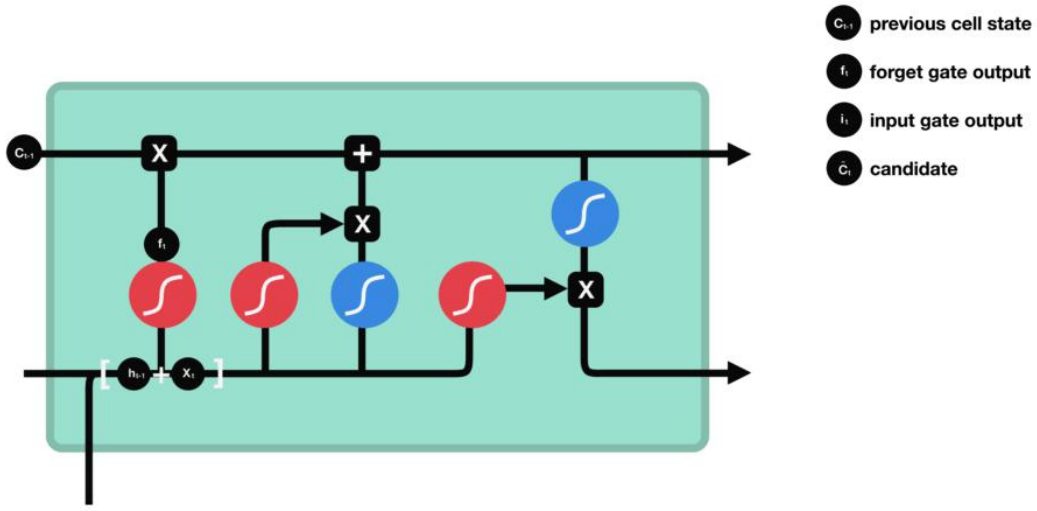
The core concept of LSTM's are the cell state, and it's various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it's way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information get's added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.
there are two state value(cell and hiden in each state)

**Forget Gate**

This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

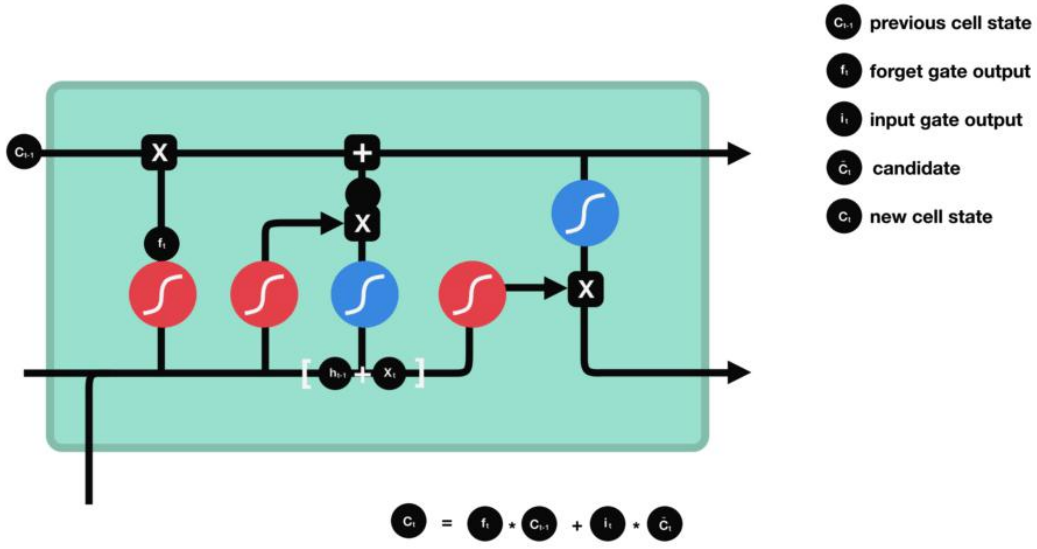
**Input Gate**

First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.



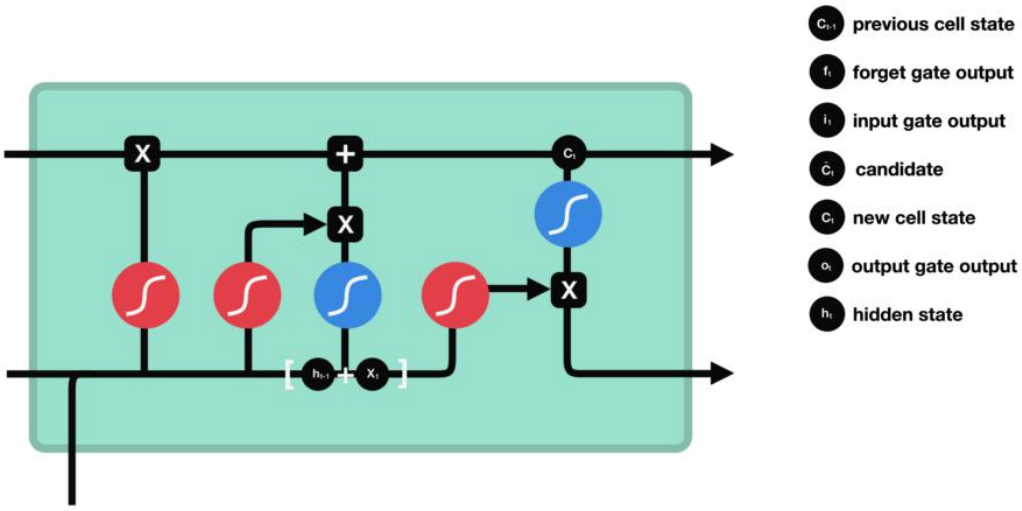

**Cell Gate**

First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

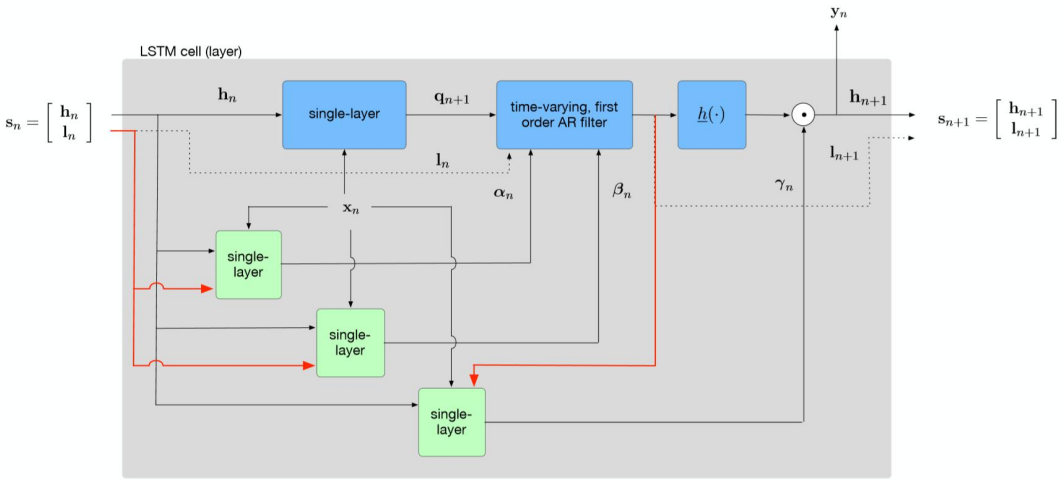



**Output Gate**

The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.




**OTHER VERSION(Keras)**




$$\boldsymbol{\alpha}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\alpha \mathbf{h}_n + \mathbf{P}_\alpha \mathbf{l}_n + \mathbf{W}_\alpha \mathbf{x}_n + \mathbf{b}_\alpha)$$

$$\boldsymbol{\beta}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\beta \mathbf{h}_n + \mathbf{P}_\beta \mathbf{l}_n + \mathbf{W}_\beta \mathbf{x}_n + \mathbf{b}_\beta)$$

$$\boldsymbol{\gamma}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\gamma \mathbf{h}_n + \mathbf{P}_\gamma \mathbf{l}_{n+1} + \mathbf{W}_\gamma \mathbf{x}_n + \mathbf{b}_\gamma)$$

(this is an experimental layer type in keras)

*or sometimes*

$$\boldsymbol{\alpha}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\alpha \mathbf{h}_n + \mathbf{p}_\alpha \odot \mathbf{l}_n + \mathbf{W}_\alpha \mathbf{x}_n + \mathbf{b}_\alpha)$$

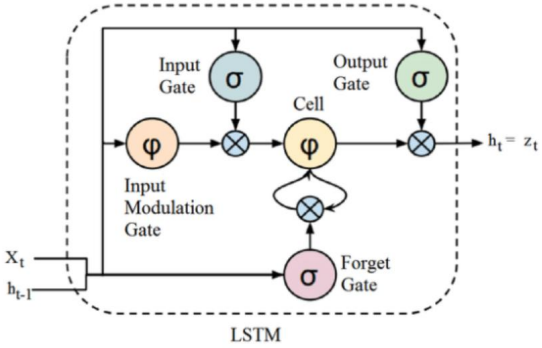
$$\boldsymbol{\beta}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\beta \mathbf{h}_n + \mathbf{p}_\beta \odot \mathbf{l}_n + \mathbf{W}_\beta \mathbf{x}_n + \mathbf{b}_\beta)$$

$$\boldsymbol{\gamma}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\gamma \mathbf{h}_n + \mathbf{p}_\gamma \odot \mathbf{l}_{n+1} + \mathbf{W}_\gamma \mathbf{x}_n + \mathbf{b}_\gamma)$$

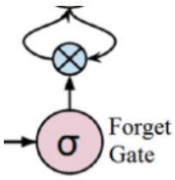
the peep-hole matrices are diagonal

**My understanding:** In LSTM, each gate has a group of parameters, so four gates, four group of patameters, each group will generate an output, and they combine together to generate the final output and hiden state so the Keras version will use more parameters (4*n*(m+n+1))

**other version of intorduction ()**



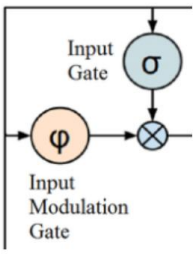

LSTM Cell




Forget Gate

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_t\right)$$

Forget Gate Equation



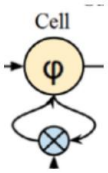

Input Gate

$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i\right)$$

Input Gate Equation

$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c\right)$$

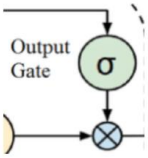
Input Modulation Gate Equation




Cell State

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t$$

Cell State Equation

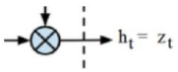



Output Gate

$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o\right)$$

Output Gate Equation

The **focus vector** is usually called the **output gate**. Out of all the possible values from the matrix, which should be moving forward to the next hidden state?




Hidden State

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

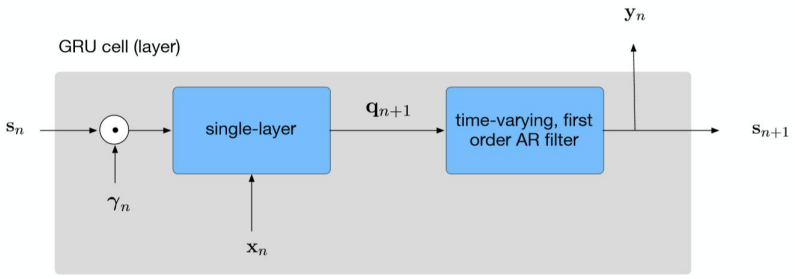
Hidden State Equation

LSTM code:

usful source: [Illustrated Guide to LSTM's and GRU's: A step by step explanation]
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9
http://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://medium.com/@kangeugine/long-short-term-memory-lstm-concept-cb3283934359#:~:text=This%20allows%20information%20from%20previous,with%20in%20the%20LSTM%20cell.&text=These%20gates%20determine%20which%20information,of%20%5B0%2C1%5D.

## GRU

# Gated Recurrent Unit (GRU)



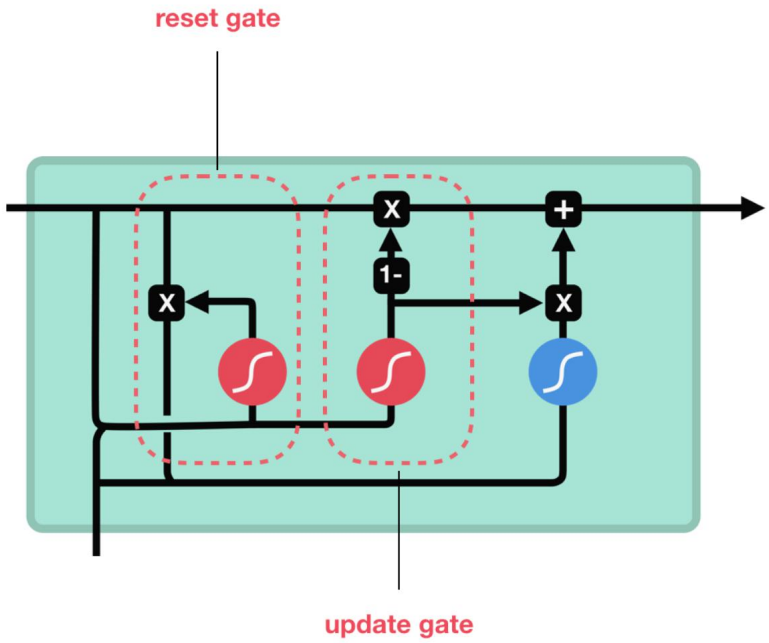

$$\mathbf{q}_{n+1} = \underline{h}(\mathbf{V}[\boldsymbol{\gamma}_n \odot \mathbf{s}_n] + \mathbf{W}\mathbf{x}_n + \mathbf{b}) \qquad \mathbf{s}_{n+1} = \boldsymbol{\alpha}_n \odot \mathbf{s}_n + (\mathbf{1} - \boldsymbol{\alpha}_n) \odot \mathbf{q}_{n+1}$$

$\boldsymbol{\gamma}_n$ "read gate" — attenuation on current state when read for update

$\boldsymbol{\alpha}_n$

$\boldsymbol{\beta}_n = \mathbf{1} - \boldsymbol{\alpha}_n$

"forget gate" and "write gate" coupled ~ unit DC gain filtering




**Update gate**

(samilar to foget+input gate)

$$\mathbf{z}_t = \underline{\sigma}(\mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{W}_z\mathbf{x}_t + \mathbf{b}_z)$$ "update gate" ("forget gate")

**Reset Gate**

(samiar to )

$$\mathbf{r}_t = \underline{\sigma}(\mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{W}_r \mathbf{x}_t + \mathbf{b}_r)$$ "regret gate" ("read gate")

$$\tilde{\mathbf{h}}_t = \underline{h}(\mathbf{U}[\mathbf{r}_t \odot \mathbf{h}_{t-1}] + \mathbf{W}\mathbf{x}_t + \mathbf{b})$$

$$\mathbf{h}_t = \mathbf{z}_n \odot \mathbf{h}_{t-1} + (\mathbf{1} - \mathbf{z}_n) \odot \tilde{\mathbf{h}}_t$$

## GRU Trainable Parameters

$$\boldsymbol{\alpha}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\alpha \mathbf{s}_n + \mathbf{W}_\alpha \mathbf{x}_n + \mathbf{b}_\alpha)$$

$$\boldsymbol{\gamma}_n = \underline{h}_{\mathrm{r}}(\mathbf{V}_\gamma \mathbf{s}_n + \mathbf{W}_\gamma \mathbf{x}_n + \mathbf{b}_\gamma)$$

$$\mathbf{q}_{n+1} = \underline{h}(\mathbf{V}[\boldsymbol{\gamma}_n \odot \mathbf{s}_n] + \mathbf{W}\mathbf{x}_n + \mathbf{b})$$

$$\mathbf{s}_{n+1} = \boldsymbol{\alpha}_n \odot \mathbf{s}_n + \boldsymbol{\beta}_n \odot \mathbf{q}_{n+1}$$

$$\mathbf{s}_{n+1} = \boldsymbol{\alpha}_n \odot \mathbf{s}_n + (\mathbf{1} - \boldsymbol{\alpha}_n) \odot \mathbf{q}_{n+1}$$

$$\mathbf{y}_n = \mathbf{s}_{n+1}$$

| | |
|---|---|
| **V** : N x N | **3** |
| **W** : N x M | **3** |
| **b** : N x 1 | **3** |

Number of Trainable Parameters: $3N(N + M + 1)$

## Back-Propagation through Time (BPTT)

## Word Embedding

### BERT

BERT (Bidirectional Encoder Representations from Transformers), released in late 2018, is a method to **pretrain language representations** that was used to create models that NLP practicioners can then download and use for free. You can either use these models to extract high quality language features from your text data, or you can fine-tune these models on a specific task (classification, entity recognition, question answering, etc.) with your own data to produce state of the art predictions.

GRT-3

# TensorFlow and Keras

## sequential model

# Pytorch