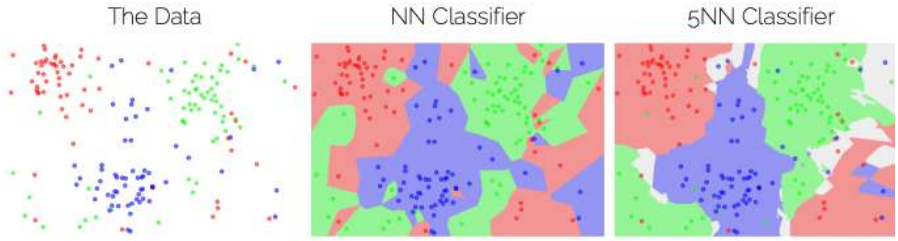
NOTES INTRO TO DEEP LEARNING

LECTURE 2: Machin Learning Basics

A Simple Classifier

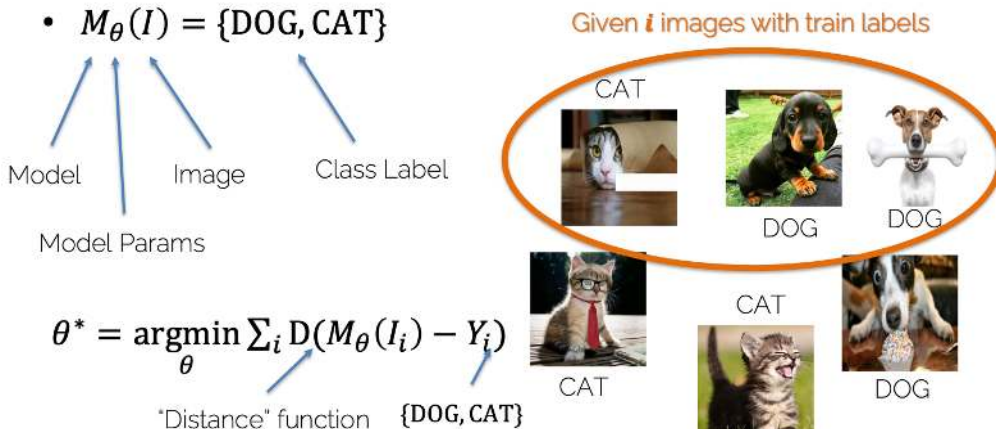
Nearest Neighbor



- Hyperparameters —> L1 distance $|x - c|$, L2 distance $||x - c||_2$, No of Neighbors: k
- These parameters are problem dependent.

Machine Learning Classification

How can we learn to perform image classification—> By experience
If we want to classify several images of dogs and cats:



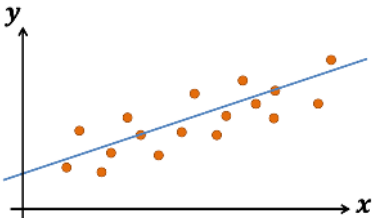
In machine learning we split our data in three sets, the train set (60%) the validation set (20%) and the test set (20%). In the first set we find the model parameters θ and in the join of the first and the second we find the hyperparameters. The test sei is only used once. Other splits are also possible e.g., 80% 10% 10%.

How can we learn to perform image classification ? Our task is to classify images, our performance measure is accuracy and our experience is the data.

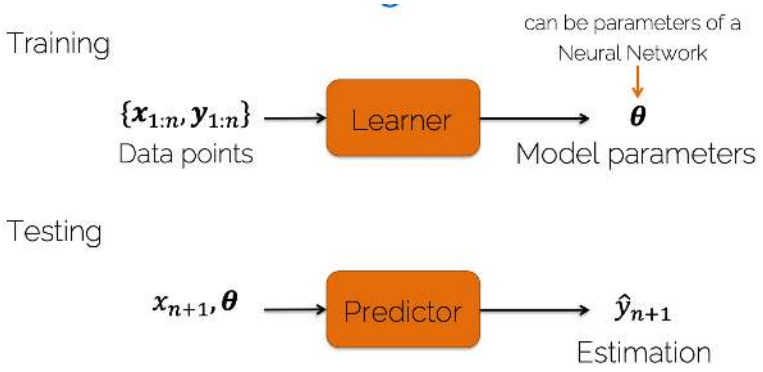
We have to types of learnings: Unsupervised learning that has no label or target class, find out properties of the structure of the data uses clustering (k-means, PCA, etc.) and Supervised learning that has labels or target classes.

Linear Regression

Is a supervised learning, the objective is to find a linear model that explains a target y given inputs x

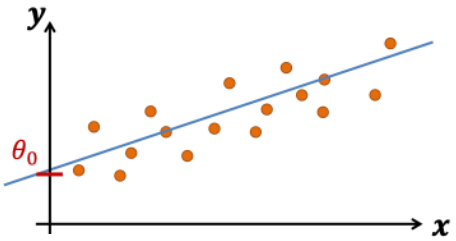


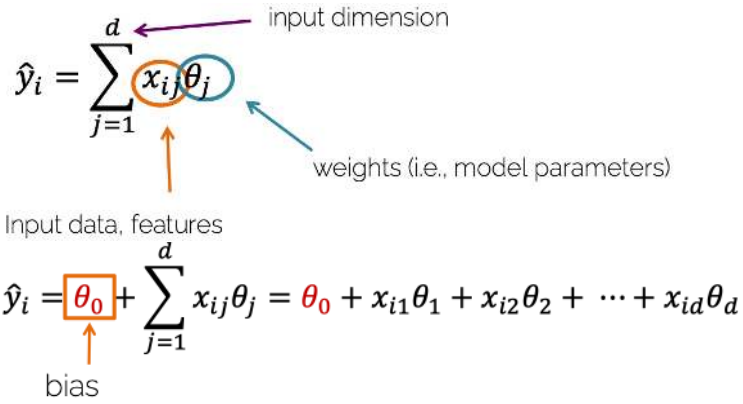
The x data points are the Input (e.g. image measurement) and the y data points are the labels (e.g cat/dog)



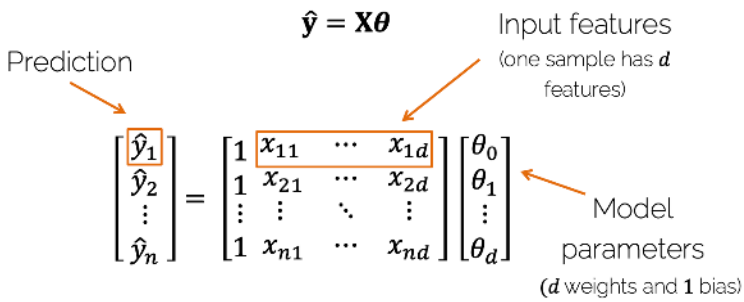
Linear prediction

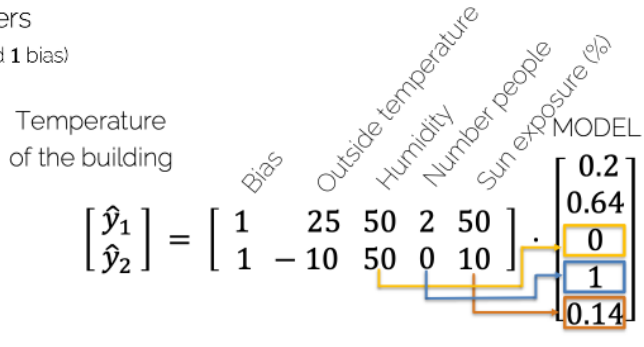
A linear model is expressed in the form



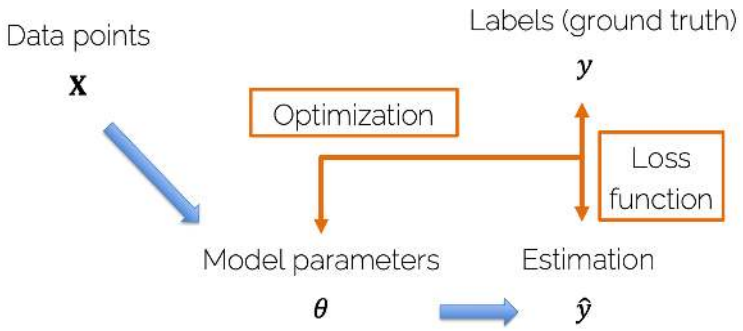


An example could be the temperature of a building. We will have x1 that will be the outside temperature (θ1), x2 that will be the level of humidity (θ2), x3 that will be the number of people (θ3) and x4 that will be the sun exposure (θ4) all these makes the temperature of a building.





How to Obtain the Model

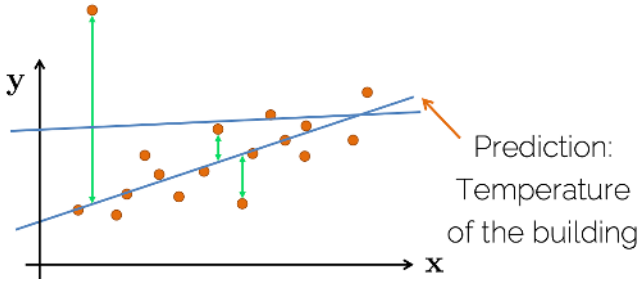


- **Loss function:** measures how good my estimation is (how good my model is) and tells the optimization method how to make it better.
- **Optimization:** changes the model in order to improve the loss function (i.e to improve my estimation).

Loss function

To minimize the error of prediction and get the objective function that is the energy cost function we use the loss function.

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$



Linear Least Squares

Linear least squares: an approach to fit a linear model to the data

Convex problem, there exists a closed-form solution that is unique.

$$\min_{\theta} J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$$\min_{\theta} J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i\boldsymbol{\theta} - y_i)^2$$

n training samples

The estimation comes from the linear model

$$\min_{\theta} J(\boldsymbol{\theta}) = (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Matrix notation

n training samples, each input vector has size d

n labels

Optimization

We have found an analytical solution to a convex problem

We think that least squares estimate is the best estimate

$$\frac{\partial J(\theta)}{\partial \theta} = 2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - 2\mathbf{X}^T\mathbf{y} = 0$$

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Inputs: Outside temperature, number of people,

True output: Temperature of the building

Maximum Likelihood

Maximum Likelihood Estimate

A method of estimating the parameters of a statistical model given observations, $p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$

Observations from $p_{data}(\mathbf{y}|\mathbf{X})$

A method of estimating the parameters of a statistical model given observations, by finding the parameters of a statistical model given observations, by finding the parameter values the maximize the likelihood of making the observations given the parameters.

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$$

MLE assumes that the training samples are independent and generated by the same probability distribution.

$$p_{model}(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}) = \prod_{i=1}^{n} p_{model}(y_i|\mathbf{x}_i,\boldsymbol{\theta})$$

"i.i.d." assumption

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p_{model}(y_i|\mathbf{x}_i,\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p_{model}(y_i|\mathbf{x}_i,\boldsymbol{\theta})$$

Back to Linear Regression

$$p(y_i|\mathbf{x}_i,\boldsymbol{\theta}) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i-\mathbf{x}_i\boldsymbol{\theta})^2}$$

Gaussian Noise

Mean

Assuming $y_i = \mathbf{x}_i\boldsymbol{\theta} + \sigma_i$ with $\sigma_i \sim \mathcal{N}(0,\sigma^2)$

$\rightarrow$ $y_i \sim \mathcal{N}(\mathbf{x}_i\boldsymbol{\theta},\sigma^2)$

Gaussian:

$$p(y_i) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}(y_i-\mu)^2}$$

$y_i \sim \mathcal{N}(\mu,\sigma^2)$

$$p(y_i|\mathbf{x}_i,\boldsymbol{\theta}) = (2\pi\sigma^2)^{-1/2} e^{-\frac{1}{2\sigma^2}(y_i-\mathbf{x}_i\boldsymbol{\theta})^2}$$

Original optimization problem

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p_{model}(y_i|\mathbf{x}_i,\boldsymbol{\theta})$$

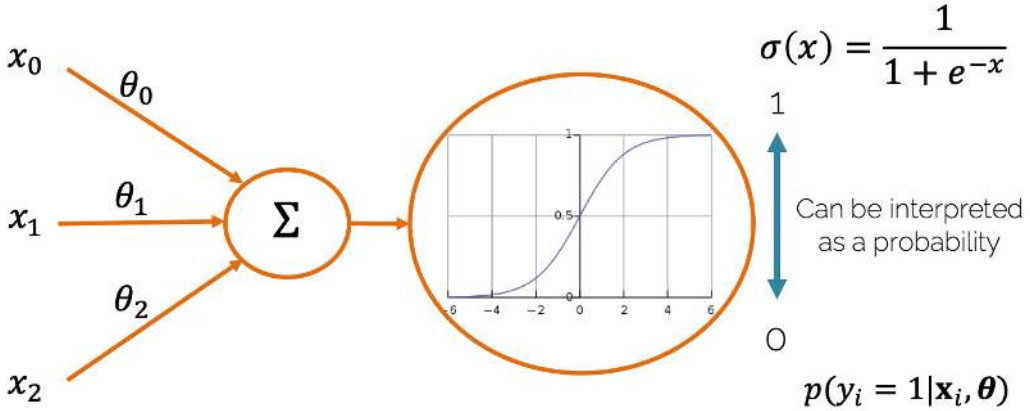
Maximum likelihood Estimate (MLE) corresponds to the Least Squares Estimate (given the assumptions).
Introduced the concepts of loss function and optimization to obtain the best model for regression

Regression vs Classification

- Regression: Predict a continuous output value (e.g. temperature of a room)
- Classification: predict a discrete value
    - Binary classification; output is either 0 or 1.
    - Multi-class classification: set of N classes.

Logistic Regression

Sigmoid for Binary Prediction



Logistic Regression

- Probability of a binary output —> $p(\mathrm{y}|\mathbf{X},\boldsymbol{\theta}) = \hat{\mathbf{y}} = \prod_{i=1}^{n} \hat{y}_i^{y_i}(1-\hat{y}_i)^{(1-y_i)}$
- Maximum Likelihood Estimate —> $\boldsymbol{\theta_{ML}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathrm{y}|\mathbf{X},\boldsymbol{\theta})$

$$p(\mathrm{y}|\mathbf{X},\boldsymbol{\theta}) = \hat{\mathbf{y}} = \prod_{i=1}^{n} \hat{y}_i^{y_i}(1-\hat{y}_i)^{(1-y_i)}$$

$$\sum_{i=1}^{n} \log\left(\hat{y}_i^{y_i}(1-\hat{y}_i)^{(1-y_i)}\right)$$

$$\sum_{i=1}^{n} y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$$

$$\mathcal{L}(\hat{y}_i, y_i) = -[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

Maximize likelihood by minimizing the loss function

$$y_i = 1 \longrightarrow \mathcal{L}(\hat{y}_i, 1) = -\log \hat{y}_i$$

Maximize! $\boldsymbol{\theta_{ML}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathrm{y}|\mathbf{X},\boldsymbol{\theta})$

- Loss function —> $\mathcal{L}(\hat{y}_i, y_i) = -[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$
- Cost function —>

$$C(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y_i)$$

Minimization

$$= -\frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$$

$\hat{y}_i = \sigma(\mathbf{x}_i\boldsymbol{\theta})$

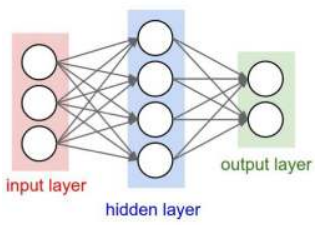
- No closed-form solution
- Make use of an iterative method —> Gradient descent

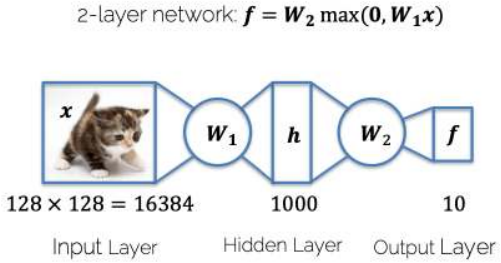
LECTURE 3: Introduction to Neural Networks

Introduction to Neural Networks

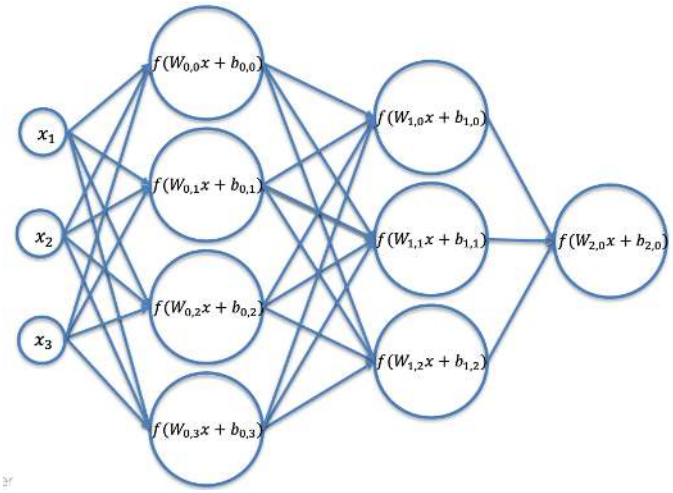
Neural Networks

Non-linear score function f = ... (max(0, $W1x$))

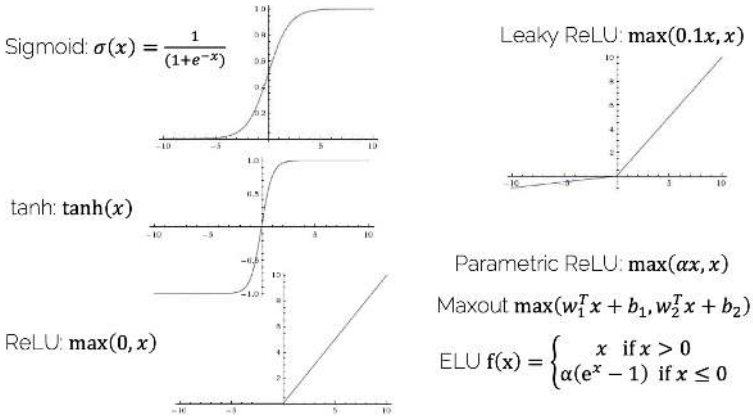




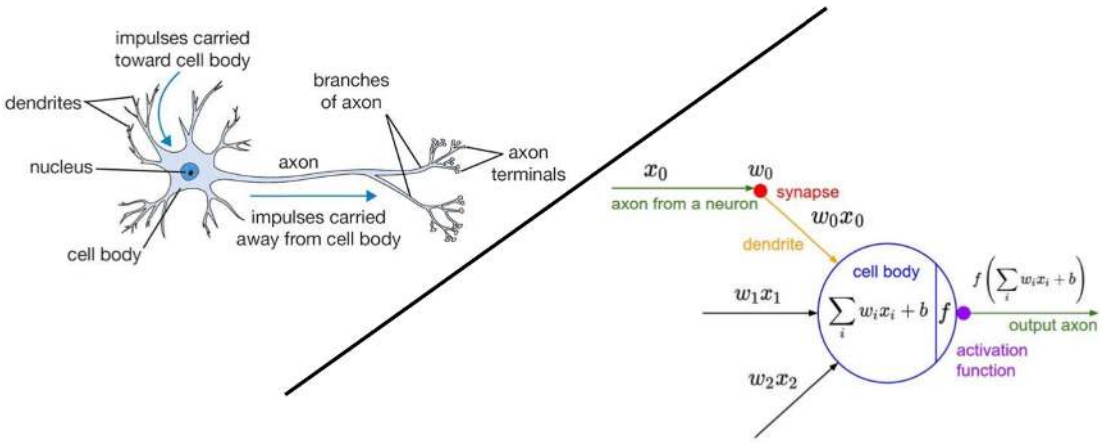
Net of Artificial Neurons



Activation Functions



Biological Neurons



Artificial Neural Networks

Artificial neural Networks are inspired by the brain, but not even close in terms of complexity! The comparison is great for the media and news articles.

Neural Network

- Given a dataset with ground truth training pairs [xi ; yi],

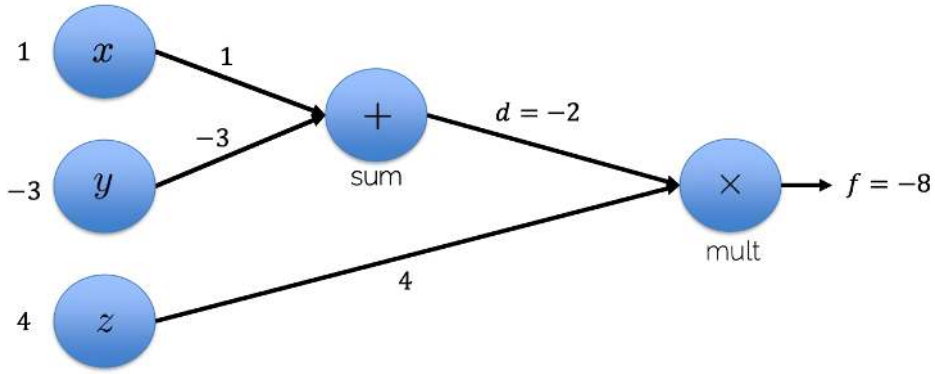
- Find optimal weights and biases $\boldsymbol{W}$ using stochastic gradient descent, such that the loss function is minimized
    - Compute gradients with back-propagation (use-batch-mode)
    - Iterate many times over training set (SGD)

Computational Graphs

Computational Graphs

- Directional graph
- Matrix operations are represented as compute nodes.
- Vertex nodes are variables or operators like, +, -, *, /, log(), exp()…
- Directional edges show flow of inputs to vertices.

- $f(x, y, z) = (x + y) \cdot z$ Initialization $x = 1, y = -3, z = 4$



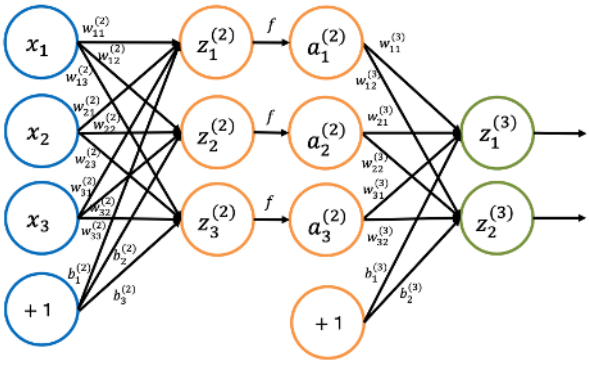
- neural networks have complicated architectures

$\boldsymbol{f}$ = $\boldsymbol{W5}\sigma(\boldsymbol{W4}$ tanh($\boldsymbol{W3}$, max($\mathbf{0}$, $\boldsymbol{W2}$ max($\mathbf{0}$, $\boldsymbol{W1x}$))))

- Lot of matrix operations!
- Represent NN as computational graph

A neural network can be represented as a computational graph…

- It has compute nodes (operations)
- It has edges that connect nodes (data flow)
- It is directional
- It can be organized into ‘layers’



$$z_k^{(2)} = \sum_i x_i w_{ik}^{(2)} + b_k^{(2)}$$

$$a_k^{(2)} = f(z_k^{(2)})$$

$$z_k^{(3)} = \sum_i a_i^{(2)} w_{ik}^{(3)} + b_k^{(3)}$$

- The computation of Neural Network has further meanings:
    - The multiplication of $\boldsymbol{W}$ and $\boldsymbol{x}$ encode input information
    - The activation function: select the key features
    - The convolutional layers: extract useful features with shared weights.

Loss Function

Loss Functions

- A function to measure the goodness of the predictions (or equivalently, the network's performance)

Intuitively,…

- A large loss indicates bad predictions/performance (—> performance needs to be improved by training the model)
- The choice of the loss function depends on the concrete problem or the distribution of the target variable.

Regression Loss

- L1 Loss:

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}};\ \boldsymbol{\theta}) = \frac{1}{n}\sum_{i}^{n} ||y_i\ - \hat{y}_i||_1$$

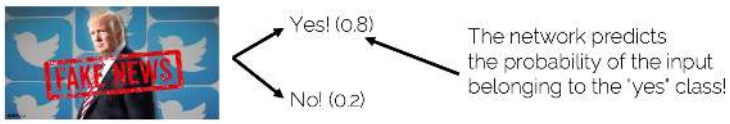
- MSE Loss:

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}};\ \boldsymbol{\theta}) = \frac{1}{n}\sum_{i}^{n} ||y_i\ - \hat{y}_i||_2^2$$

Binary Cross Entropy

- Loss function for binary (yes/no) classification

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}};\ \boldsymbol{\theta}) = -\frac{1}{n}\sum_{i}^{n} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$



Cross Entropy

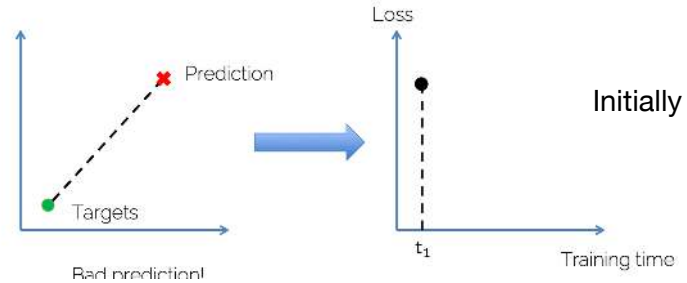
- loss function for multi-class classification

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}};\ \boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{k=1}^{k} (y_{ik} \cdot \log \hat{y}_{ik})$$

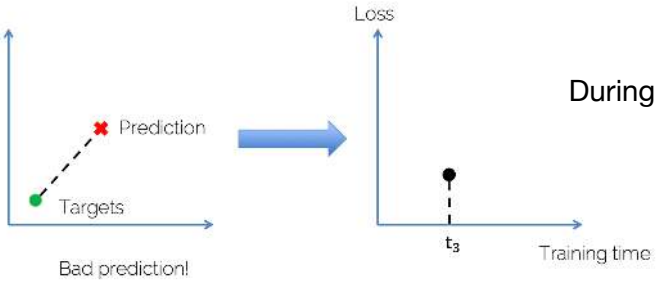
This generalizes the binary case from the slide before!

More General case

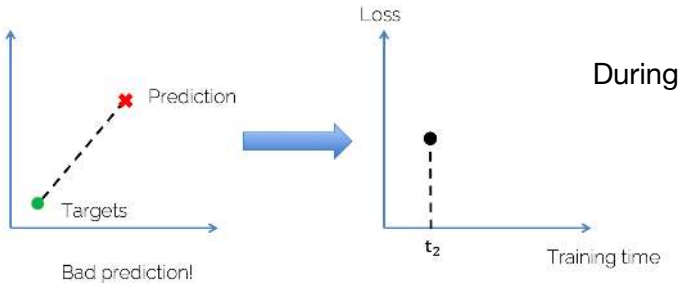
- Ground truth: $\boldsymbol{y}$
- Prediction: $\hat{y}$
- Loss function: $L(\boldsymbol{y}, \hat{y})$
- Motivation:
  - Minimize the loss <—> find better predictions
  - Predictions are generated by the NN
  - Find better predictions <—> find better NN



Initially

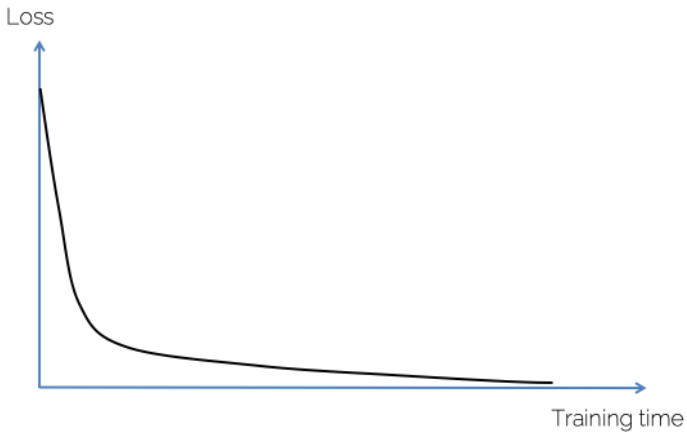


During Training…

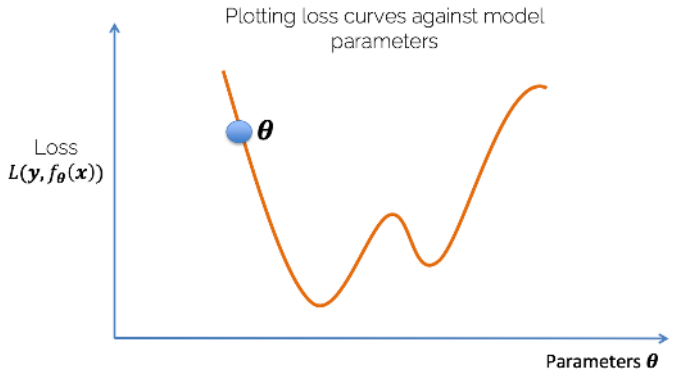


During Training…

Training Curve

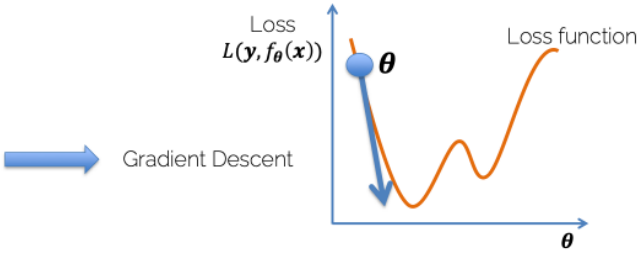


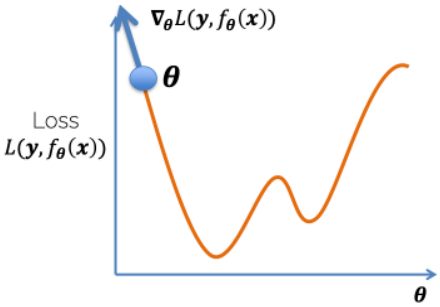
How to find a better NN?



-Loss function: $L(\boldsymbol{y},\hat{y}) = L(\boldsymbol{y}, f\ \theta(\ \boldsymbol{x})\ )$
-Neural Network: $f\theta(\boldsymbol{x})$
-Goal: minimize the loss w.r.t θ

- Minimize $L(\boldsymbol{y}, f\ \theta(\ \boldsymbol{x})\ )$ w.r.t. θ
- In the context of NN, we use gradient-based optimization



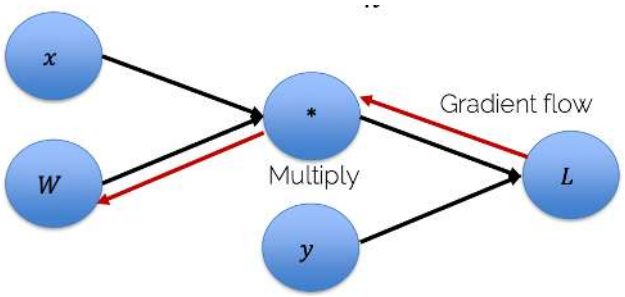


Learning rate

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}))$$

$$\boldsymbol{\theta}^* = \arg\min L(\boldsymbol{y}, f_{\boldsymbol{\theta}}(\boldsymbol{x}))$$

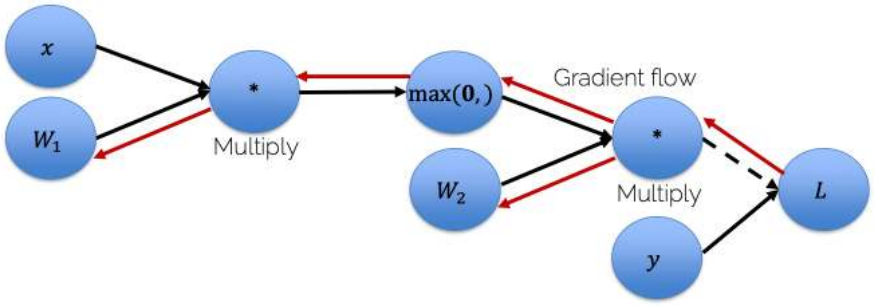
- Given inputs $\boldsymbol{x}$ and targets $\boldsymbol{y}$
- Given one layer NN with no activation function $\boldsymbol{f}\theta\ \boldsymbol{x} = \boldsymbol{W}\boldsymbol{x}$, $\theta = \boldsymbol{W}$ Later $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$
- Given MSE Loss: $L(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{\theta}) = \frac{1}{n}\sum_i^n ||y_i - \hat{y}_i||_2^2$



- $\nabla_\theta L(\boldsymbol{y}, f_\theta(\boldsymbol{x})) = \frac{2}{n}\sum_i^n(\boldsymbol{W} \cdot x_i - y_i) \cdot x_i^T$

- Given inputs $\boldsymbol{x}$ and targets $\boldsymbol{y}$
- Given a multi-layer NN with many activations $\boldsymbol{f}$ = $\boldsymbol{W5}\sigma(\boldsymbol{W4}$ tanh($\boldsymbol{W3}$, max($\boldsymbol{0}$, $\boldsymbol{W2}$ max($\boldsymbol{0}$, $\boldsymbol{W1}\boldsymbol{x}$))))
- Gradient descent for $L(\boldsymbol{y}, f\theta(\boldsymbol{x}))$ w. r. t. θ
  - Need to propagate gradients from end to first layer ($\boldsymbol{W1}$).

- Given inputs $\boldsymbol{x}$ and targets $\boldsymbol{y}$
- Given a multi-layer NN with many activations



- Back-propagation: Use chain rule to compute gradients.
  - Compute graphs come in handy!

- Why gradient descent ?
  - Easy to compute using compute graphs
- Other methods include
  - Newtons method
  - L-BFGS
  - Adaptive moments
  - Conjugate gradient

LECTURE 4: Optimization and Back-propagation

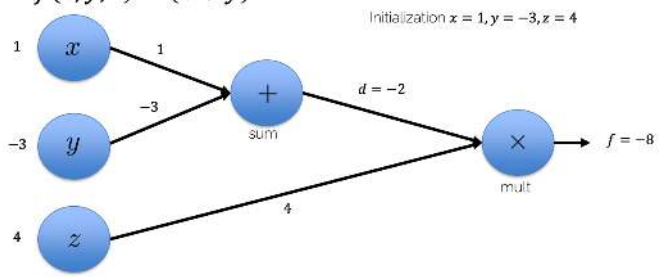
Backprop

The Importance of Gradients

- Our optimization schemes are based on computing gradients $\nabla\theta L(\theta)$
- One can compute gradients analytically but what if our function is too complex?
- Break down gradient computation Backpropagation

Backprop: Forward Pass

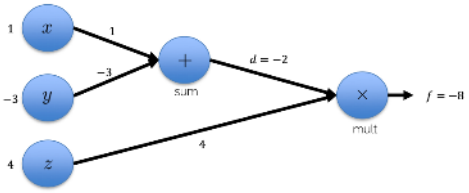
- $f(x,y,z) = (x+y)\cdot z$



Backprop: Backward Pass

$f(x,y,z) = (x+y)\cdot z$

with $x=1, y=-3, z=4$



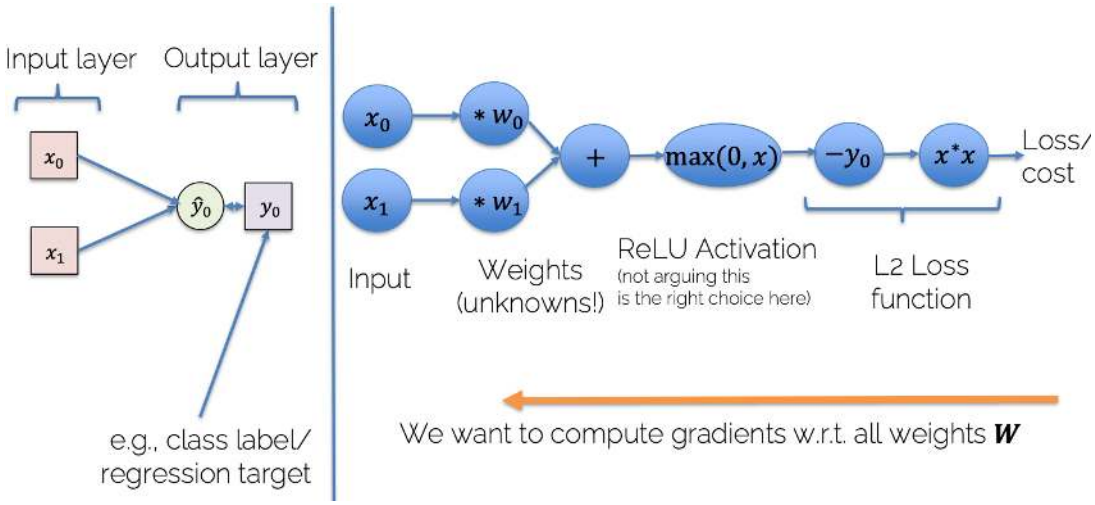
$d = x + y$ $\quad \frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$

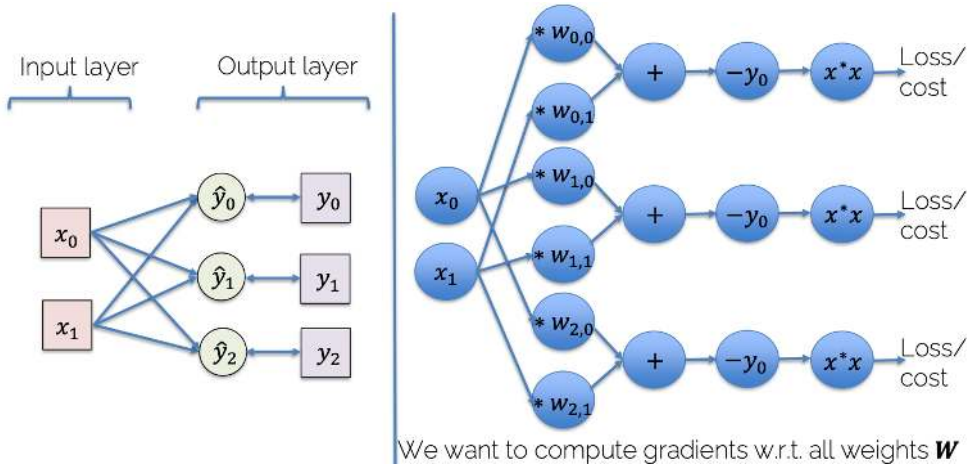
$f = d \cdot z$ $\quad \frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$

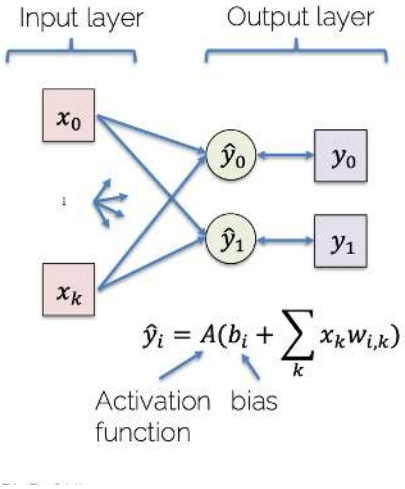
What is $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?

Compute Graphs —> Neural Networks

- xk input variables
- wl,m,n network wights(note 3 indices)
  - l which layer
  - m which neuron
  - n which weight neuron
- ŷi computed output (i output dim; $nout$)
- yi ground truth targets
- L loss function







Goal: We want to compute gradients of the loss function L w.r.t. all weights $\boldsymbol{W}$

$$L = \sum_i L_i$$

L: sum over loss per sample, e.g. L2 loss ⟶ simply sum up squares:

$$L_i = (\hat{y}_i - y_i)^2$$

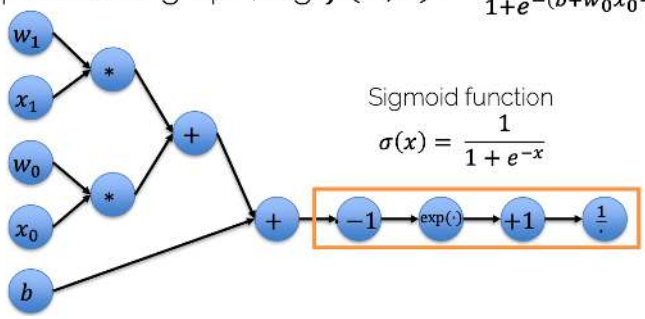
⟶ use chain rule to compute partials

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

We want to compute gradients w.r.t. all weights $\boldsymbol{W}$ AND all biases $\boldsymbol{b}$

NNs as Computational Graphs

We can express any kind of functions in a computational graph, e.g. $\boldsymbol{f(w,x)} = \frac{1}{1+e^{-(b+w_0x_0+w_1x_1)}}$



Sigmoid function

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

Gradient Descent

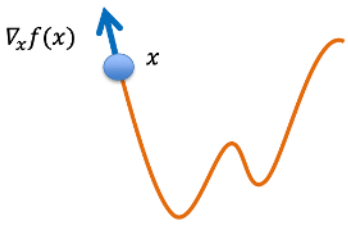
Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



$$x' = x - \alpha \nabla_x f(x)$$

Learning rate

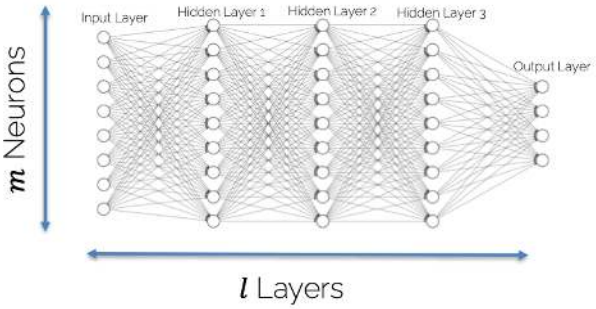
Gradient Descent for Neural Networks

For a given training pair $\{\boldsymbol{x}, \boldsymbol{y}\}$, we want to update all weights, i.e., we need to compute the derivatives w.r.t. to all weights:

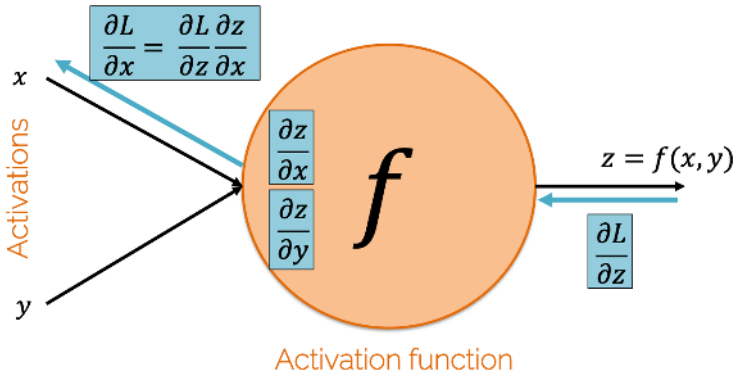
$$\nabla_W f_{\{x,y\}}(W) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \cdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{bmatrix}$$

Gradient step:

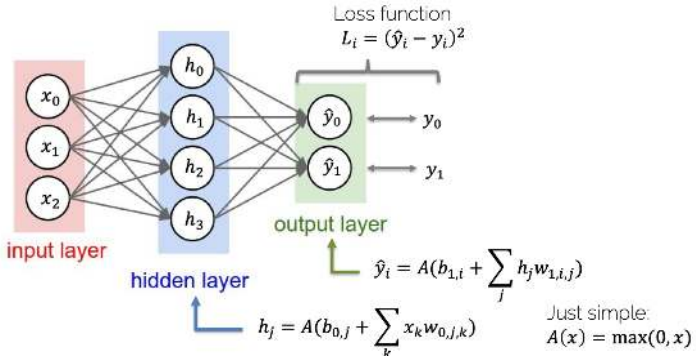
$$W' = W - \alpha \nabla_W f_{\{x,y\}}(W)$$

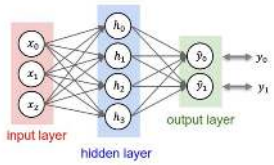


The Flow of the Gradient



Gradient descent for Neural Networks





$$h_j = A(b_{0,j} + \sum_k x_k w_{0,j,k})$$

$$\hat{y}_i = A(b_{1,i} + \sum_j h_j w_{1,i,j})$$

$$L_i = (\hat{y}_i - y_i)^2$$

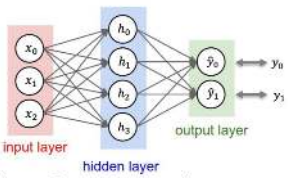
How many unknown weights?

- Output layer: $2 \cdot 4 + 2$
- Hidden Layer: $4 \cdot 3 + 4$

#neurons · #input channels + #biases

Note that some activations have also weights

Derivatives of Cross Entropy Loss



Binary Cross Entropy loss

$$L = -\sum_{i=1}^{n_{out}} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

$$\hat{y}_i = \frac{1}{1 + e^{-s_i}} \quad s_i = \sum_j h_j w_{ji}$$

output scores

Gradients of weights of last layer:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_{ji}}$$

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{-y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)},$$

$$\frac{\partial \hat{y}_i}{\partial s_i} = \hat{y}_i (1 - \hat{y}_i),$$

$$\frac{\partial s_i}{\partial w_{ji}} = h_j$$

$$\Rightarrow \frac{\partial L}{\partial w_{ji}} = (\hat{y}_i - y_i) h_j, \quad \frac{\partial L}{\partial s_i} = \hat{y}_i - y_i$$

Gradients of weights of first layer:

$$\frac{\partial L}{\partial h_j} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_i} \frac{\partial s_i}{\partial h_j} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial \hat{y}_i} \hat{y}_i (1 - \hat{y}_i) w_{ji} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji}$$

$$\frac{\partial L}{\partial s_j^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_i} \frac{\partial s_i}{\partial h_j} \frac{\partial h_j}{\partial s_j^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji} (h_j (1 - h_j))$$

$$\frac{\partial L}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} \frac{\partial L}{\partial s_j^1} \frac{\partial s_j^1}{\partial w_{kj}^1} = \sum_{i=1}^{n_{out}} (\hat{y}_i - y_i) w_{ji} \left(h_j (1 - h_j)\right) x_k$$

Gradient Descent for Neural Networks

Initialize $x = 1,\ y = 0,$
$w_1 = \frac{1}{3}, w_2 = 2$

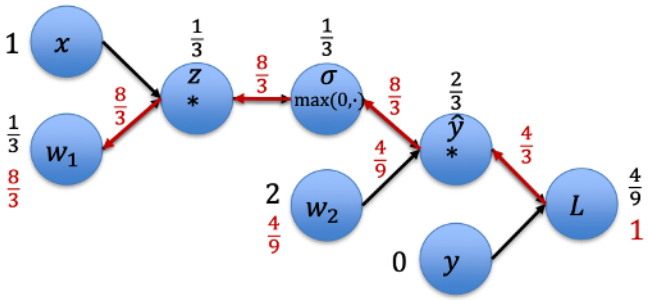
In our case $n, d = 1$:

$L = (\hat{y} - y)^2 \Rightarrow \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$

$\hat{y} = w_2 \cdot \sigma \Rightarrow \frac{\partial \hat{y}}{\partial \sigma} = w_2$

$\sigma = \max(0, z) \Rightarrow \frac{\partial \sigma}{\partial z} = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ else} \end{cases}$

$z = x \cdot w_1 \Rightarrow \boxed{\frac{\partial z}{\partial w_1} = x}$



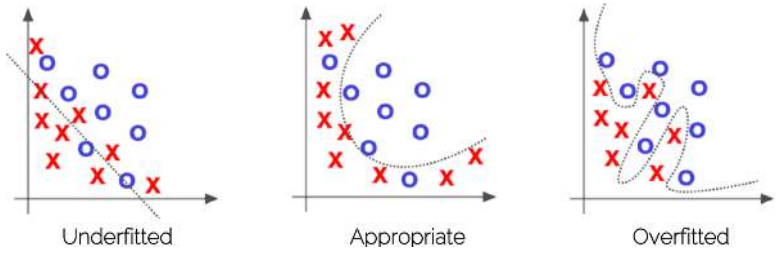
Backpropagation

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$2 \cdot \frac{2}{3} \cdot 2 \cdot 1 \cdot 1$

Regularization

Over- and Under-fitting



Regularization

Loss function $L(\boldsymbol{y}, \hat{\boldsymbol{y}}, \boldsymbol{\theta}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda R(\boldsymbol{\theta})$

Regularization techniques
- L2 regularization
- L1 regularization

Add regularization term to loss function

- Max norm regularization
- Dropout
- Early stopping
- ...

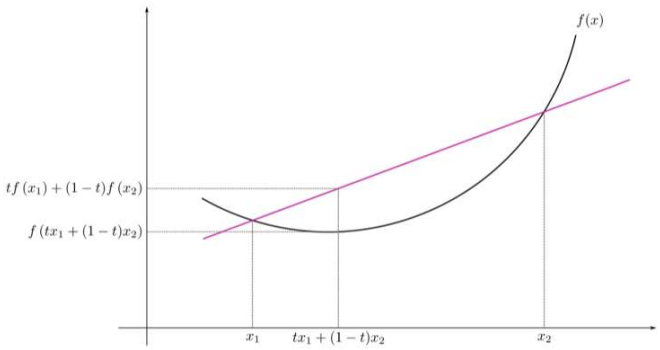
More details later

LECTURE 5: Scaling Optimization

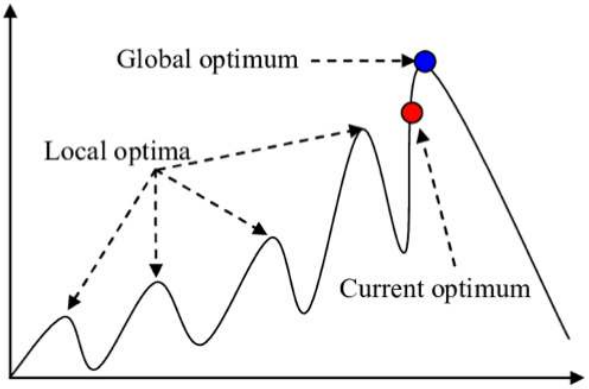
Optimization

Convergence of Gradient Descent

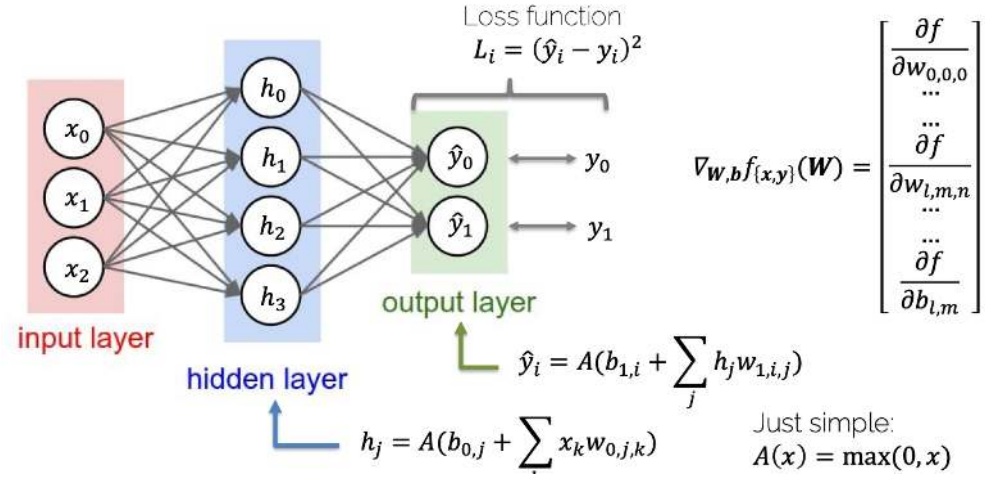
- Convex function: all local minima are global minima



- Neural networks are non-convex
  - Many (different) local minima
  - No (practical) way to say which is globally optimal

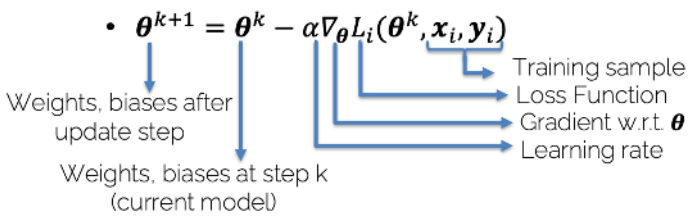


Gradient Descent for Neural Networks



Gradient Descent: Single Training Sample

- Given a loss function L and a single training sample $\{\boldsymbol{xi}, \boldsymbol{yi}\}$
- Find best model parameters $\theta = \{\boldsymbol{W}, \boldsymbol{b}\}$
- Cost $Li(\theta, \boldsymbol{xi}, \boldsymbol{yi})$
  - $\theta = \arg\min Li(\boldsymbol{xi}, \boldsymbol{yi})$
- Gradient Descent:
  - Initialize θ^1 with 'random' values (more on that later)
  - $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i)$
  - Iterate until convergence: $|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k| < \epsilon$



- $\nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i)$ computed via backpropagation
- Typically: $\dim\left(\nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i)\right) = \dim(\boldsymbol{\theta}) \gg 1\ million$

Gradient Descent: Multiple Training Samples

- Given a loss function L and multiple (n) training samples $\{xi, yi\}$
- Find best model parameters $\theta = \{W, b\}$
- Cost $L = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)$
  - $\theta = \text{argmin} L$
- Update step for multiple samples $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k, \boldsymbol{x}_{\{1..n\}}, \boldsymbol{y}_{\{1..n\}}\right)$
- Gradient is average/ sum over residuals

$$\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^k, \boldsymbol{x}_{\{1..n\}}, \boldsymbol{y}_{\{1..n\}}\right) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\nabla_{\boldsymbol{\theta}} L_i\left(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i\right)}$$

Reminder: this comes from backprop.

- Often people are lazy and just write $\nabla L = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i$
  - Omitting 1/n is not 'wrong', it just mean rescaling the learning rate.

Side Note: Optimal Learning Rate

Can compute optimal learning rate α using Line Search (optimal for a given set)

1. Compute gradient: $\nabla_{\boldsymbol{\theta}} L = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i$
2. Optimize for optimal step $\boldsymbol{\alpha}$:

$$\underbrace{\arg\min_{\alpha} L(\boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L)}_{\boldsymbol{\theta}^{k+1}}$$

3. $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L$

Not that practical for DL since we need to solve huge system every step..

Gradient Descent on Train Set

- Given a large train set with n training samples $\{xi\ ,\ yi\ \}$
  - Let's say 1 million labeled images
  - Let's say our network has 500k parameters
- Gradient has 500k dimension
- $n = 1\ million$

—> Extremely expensive to compute

Stochastic Gradient Descent (SGD)

- If we have n training samples, we need to compute the gradient for all of them which is $O(n)$
- If we consider the problem as empirical risk minimization, we can express the total loss over the training data as the expectation of all the samples

$$\frac{1}{n}\left(\sum_{i=1}^{n} L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)\right) = \mathbb{E}_{i\sim[1,\dots,n]}[L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)]$$

- The expectation can be approximated with a small subset of the data

$$\mathbb{E}_{i\sim[1,\dots,n]}[L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)] \approx \frac{1}{|S|}\sum_{j\in S}\left(L_j(\boldsymbol{\theta}, \boldsymbol{x}_j, \boldsymbol{y}_j)\right) \text{ with } S \subseteq \{1, \dots, n\}$$

Minibatch
choose subset of trainset $\boldsymbol{m} \ll \boldsymbol{n}$

$$B_i = \{\{\boldsymbol{x}_1, \boldsymbol{y}_1\}, \{\boldsymbol{x}_2, \boldsymbol{y}_2\}, \dots, \{\boldsymbol{x}_m, \boldsymbol{y}_m\}\}$$
$$\{B_1, B_2, \dots, B_{n/m}\}$$

- Minibatch size is hyperparameter
  - Typically power of 2 —> 8, 16, 32, 64, 128…
  - Smaller batch size means greater variance in the gradients —> noisy updates
  - Mostly limited by GPU memory (in backward pass)

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k, x_{\{1..m\}}, y_{\{1..m\}})$$

k now refers to k-th iteration

$$\nabla_\theta L = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L_i$$

m training samples in the current minibatch

Gradient for the k-th minibatch

Convergence of SGD

Suppose we want to minimize the function $F(\theta)$ with the stochastic approximation $\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$

Where $\alpha 1$, $\alpha 2$ ... αn is a sequence of positive step-size and $H(\theta k, X)$ is the unbiased estimate of $\nabla F(\theta \text{^k})$, i.e. $\mathbb{E}[H(\theta k, X)] = \nabla F(\theta \text{^k})$

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$$

converges to a local (global) minimum if the following conditions are met:

1) $\alpha_n \geq 0, \forall\, n \geq 0$
2) $\sum_{n=1}^{\infty} \alpha_n = \infty$
3) $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$
4) $F(\theta)$ is strictly convex

The proposed sequence by Robbins and Monro is $\alpha_n \propto \frac{\alpha}{n}, for\ n > 0$

Problems of SGD

- Gradient is scaled equally across all dimensions
  —> i.e., cannot independently scale directions
  —> need to have conservative min learning rate to avoid divergence.
  —> slower than 'necessary'

- Finding good learning rate is an art by itself

Gradient Descent with Momentum

$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \nabla_\theta L(\theta^k)$$

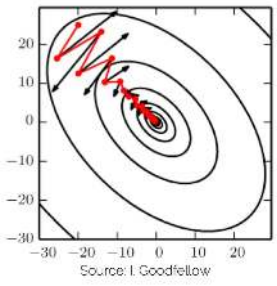
accumulation rate ('friction', momentum) — velocity — learning rate — Gradient of current minibatch

$$\theta^{k+1} = \theta^k + v^{k+1}$$

weights of model — velocity

Exponentially-weighted average of gradient

Important: velocity v^k is vector-valued!



Source: I. Goodfellow

Step will be largest when a sequence of gradients all point to the same direction

Hyperparameters are α, β

β is often set to 0.9

$$\theta^{k+1} = \theta^k + v^{k+1}$$

Nesterov Momentum

- Look-ahead momentum

$$\tilde{\boldsymbol{\theta}}^{k+1} = \boldsymbol{\theta}^k + \beta \cdot \boldsymbol{v}^k$$

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\tilde{\boldsymbol{\theta}}^{k+1})$$

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

Nesterov, Yurii E. "A method for solving the convex programming problem with convergence rate O (1/k^ 2)." *Dokl. akad. nauk Sssr.* Vol. 269. 1983.

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.

brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Source: G. Hinton

$$\tilde{\boldsymbol{\theta}}^{k+1} = \boldsymbol{\theta}^k + \beta \cdot \boldsymbol{v}^k$$
$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\tilde{\boldsymbol{\theta}}^{k+1})$$
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

Root Mean Squared Prop (RMSProp)

- RMSProp divides the learning rate by an exponentially-decaying average of squared gradients.

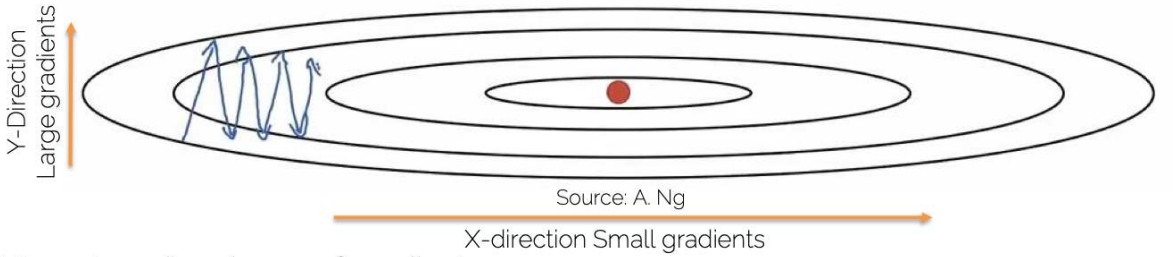
$$\boldsymbol{s}^{k+1} = \beta \cdot \boldsymbol{s}^k + (1-\beta)[\nabla_{\boldsymbol{\theta}} L \circ \nabla_{\boldsymbol{\theta}} L]$$

Element-wise multiplication

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\nabla_{\boldsymbol{\theta}} L}{\sqrt{\boldsymbol{s}^{k+1}} + \epsilon}$$

Hyperparameters: α, β, ϵ

Needs tuning! Often 0.9 Typically 10^{-8}



(Uncentered) variance of gradients
→ second momentum

$$\boldsymbol{s}^{k+1} = \beta \cdot \boldsymbol{s}^k + (1-\beta)[\nabla_{\boldsymbol{\theta}} L \circ \nabla_{\boldsymbol{\theta}} L]$$

We're dividing by square gradients:
- Division in Y-Direction will be large
- Division in X-Direction will be small

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\nabla_{\boldsymbol{\theta}} L}{\sqrt{\boldsymbol{s}^{k+1}} + \epsilon}$$

Can increase learning rate!

- Dampening the oscillations for high-variance directions.
- Can use faster learning rate because it is less likely to diverge.
  —> Speed up learning speed
  —> Second moment

Adaptive Moment Estimation (Adam)

Idea : Combine Momentum and RMSProp

$$m^{k+1} = \beta_1 \cdot m^k + (1-\beta_1)\nabla_\theta L(\theta^k)$$ ← First momentum: mean of gradients

$$v^{k+1} = \beta_2 \cdot v^k + (1-\beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)]$$ ← Second momentum: variance of gradients

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{m^{k+1}}{\sqrt{v^{k+1}}+\epsilon}$$

Note : This is not the update rule of Adam

Q. What happens at $k = 0$?
A. We need bias correction as $m^0 = 0$ and $v^0 = 0$

Adam: Bias Corrected

- Combines Momentum and RMSProp

$$m^{k+1} = \beta_1 \cdot m^k + (1-\beta_1)\nabla_\theta L(\theta^k) \qquad v^{k+1} = \beta_2 \cdot v^k + (1-\beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)$$

- m^k and v^k are initialized with zero
  —> bias towards zero
  —> Need bias-corrected moment updates

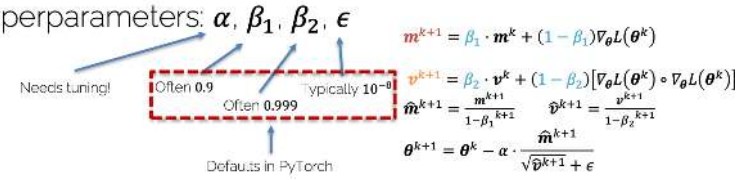
Update rule of Adam

$$\hat{m}^{k+1} = \frac{m^{k+1}}{1-\beta_1{}^{k+1}} \qquad \hat{v}^{k+1} = \frac{v^{k+1}}{1-\beta_2{}^{k+1}} \longrightarrow \theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}}+\epsilon}$$

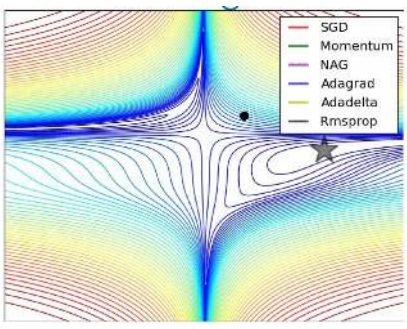
Adam

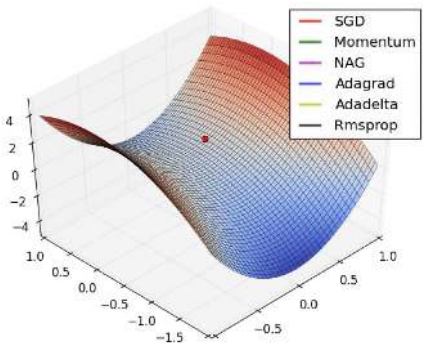
- Exponentially-decaying mean and variance of gradients (combines first and second order momentum)

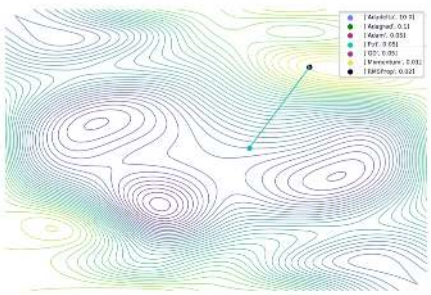
- Hyperparameters: α, β_1, β_2, ϵ



Convergence







Jacobian and Hessian

- Derivative $f: \mathbb{R} \to \mathbb{R}$ $\frac{df(x)}{dx}$
- Gradient $f: \mathbb{R}^m \to \mathbb{R}$ $\nabla_x f(\boldsymbol{x})$ $\left(\frac{df(\boldsymbol{x})}{dx_1}, \frac{df(\boldsymbol{x})}{dx_2}\right)$
- Jacobian $f: \mathbb{R}^m \to \mathbb{R}^n$ $\mathbf{J} \in \mathbb{R}^{n \times m}$
- Hessian $f: \mathbb{R}^m \to \mathbb{R}$ $\mathbf{H} \in \mathbb{R}^{m \times m}$ SECOND DERIVATIVE

Newton's Method

- Approximate our function by a second-order Taylor series expansion

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

First derivative — Second derivative (curvature)

- Differentiate and equate to zero

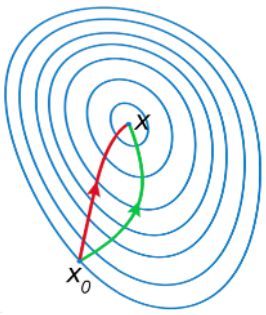
$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$ Update step

We got rid of the learning rate!

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$ Update step

Parameters of a network (millions)	Number of elements in the Hessian	Computational complexity of 'inversion' per iteration
k	k^2	$\mathcal{O}(k^3)$

SGD $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \mathbf{x}_i, \mathbf{y}_i)$



- Gradient Descent (green)
- Newton's method exploits the curvature to take a more direct route

BFGS and L-BFGS

- Broyden-Fletcher-Goldfarb-Shanno algorithm
- Belongs to the family of quasi-Newton methods
- Have an approximation of the inverse of the Hessian

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

- BFGS $\mathcal{O}(n^2)$
- Limited memory: L-BFGS $\mathcal{O}(n)$

Gauss-Newton

- $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$
  - 'true' 2nd derivatives are often hard to obtain (e.g., numerics)
  - $H_f \approx 2 J_F^T J_F$
- Gauss-Newton (GN):

$$x_{k+1} = x_k - [2 J_F(x_k)^T J_F(x_k)]^{-1} \nabla f(x_k)$$

- Solve linear system (again, inverting a matrix is unstable):

$$2\left(J_F(x_k)^T J_F(x_k)\right)(x_k - x_{k+1}) = \nabla f(x_k)$$

Solve for delta vector

Levenberg

- Levenberg
  - "damped" version of Gauss-Newton:
  - $(J_F(x_k)^T J_F(x_k) + \lambda \cdot I) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$ Tikhonov regularization
  - The damping factor λ is adjusted in each iteration ensuring:
    $$f(x_k) > f(x_{k+1})$$
    - if the equation is not fulfilled increase λ
    - →Trust region
- → "Interpolation" between Gauss-Newton (small λ) and Gradient Descent (large λ)

Levenberg-Marquardt

- Levenberg-Marquardt (LM)

$$(J_F(x_k)^T J_F(x_k) + \lambda \cdot diag(J_F(x_k)^T J_F(x_k))) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$$

  - Instead of a plain Gradient Descent for large λ, scale each component of the gradient according to the curvature.
    - Avoids slow convergence in components with a small gradient

Which, What, and When?

- Standard: Adam
- Fallback option: SGD with momentum
- Newton, LBFGS, GN, LM only if you can do full batch updates (doesn't work well for minibatches!!)

General Information

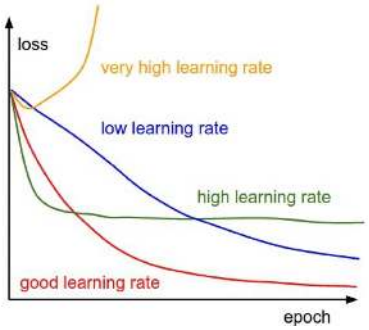
- Linear Systems (Ax = b)
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, etc.
- Non-Linear (gradient-based)
  - Newton, Gauss-Newton, LM, (L)BFGS <— second order.
  - Gradient Descent, SGD <— first order
  - Others
    - Genetic algorithm, MCMC, Metropolis-Hastings, etc.
    - Constrained and convex solvers (Langrage, ADMM, Primal-Dual, etc.)

Please Remember

- Think about your problem and optimization at hand
- SGD is specifically designed for minibatch
- When you can, use 2nd order method —> It's just faster
- GD or SGD is not a way to solve a linear system!

LECTURE 6: Training Neural Networks I

Training Neural Networks



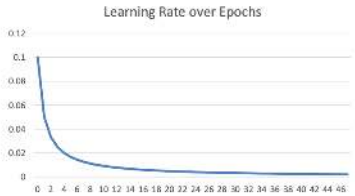
Learning Rate

- What if too high ?
- What if too low?

Need high learning rate when far away.
Need low c rate when close.

Learning Rate Decay

- $\alpha = \frac{1}{1+decay\_rate*epoch} \cdot \alpha_0$
  - E.g., $\alpha_0 = 0.1$, $decay\_rate = 1.0$
    - → Epoch 0: 0.1
    - → Epoch 1: 0.05
    - → Epoch 2: 0.033
    - → Epoch 3: 0.025
    - ...



Many options:
· Step decay $\alpha = \alpha - t \cdot \alpha$ (only every n steps)
- T is decay rate (often 0.5)

· Exponential decay $\alpha = t$^($epoch$) $\cdot$ α0
- t is decay rate (t<1.0)

· $\alpha = \frac{t}{\sqrt{epoch}} \cdot a_0$
- t is decay rate

· etc.

Training Schedule

Manually specify learning rate for entire training process

· Manually set learning rate every n-epochs
· How?
- Trial and error (the hard way)
- Some experience (only generalizes to some degree)

Consider # epochs, training set size, network size, etc.

Basic Recipe for Training

· Given a dataset with labels
- {xi,yi}
  · xi is the ith training image, with label yi
  · Often dim $x \gg$ dim(y) (e.g., for classification)
  · i is often in the 100-thousands or millions

- Take network f and its parameters w, b
- Uses SGD (or variation) to find optimal parameters w, b
  · Gradients from back propagation

Gradient Descent on Train Set

· Given a large train set with (n) training samples $\{xi\ ,\ yi\ \}$
  - Let's say 1 million labeled images
  - Let's say our network has 500k parameters

· Gradient has 500k dimensions
· $n = 1\ million$
· Extremely expensive to compute

Learning

· Learning means generalization to unknown dataset
  - (So for no 'real' learning)
  - i.e., train on known dataset —> test with optimized parameters on unknown dataset.

· Basically, we hope that based on the train set, the optimized parameters will give similar results on different data (i.e. test data)

· Training set ('train'): Use for training your neural network

· Validation set ('val'): Hyperparameter optimization, check generalization progress

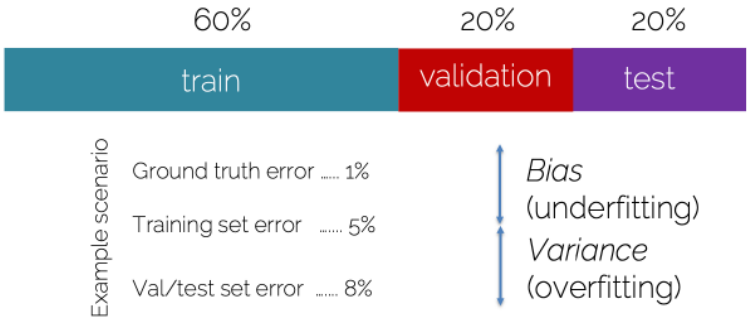
· Test set ('test'): Only for the very end, NEVER TOUCH DURING DEVELOPMENT OR TRAINING

· Typical splits:
  - Train (60%), Val (20%), Test (20%)
  - Train (80%), Val(10%), Test (10%)

· During training:
  - Train error comes from average minibatch error
  - Typically take subset of validation every n iterations.

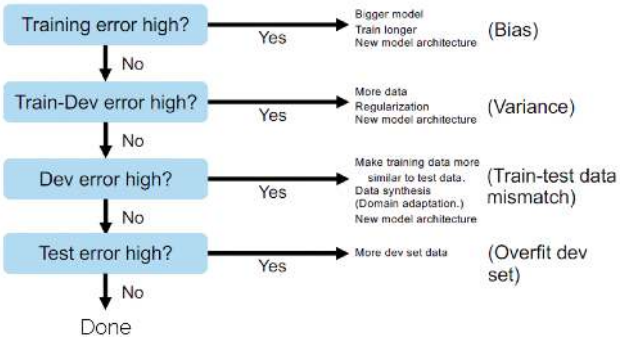
Basic Recipe for Machine Learning

· Split your data

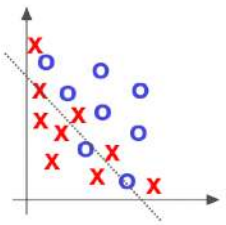


Cross Validation

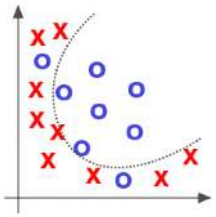
Basic Recipe for Machine Learning



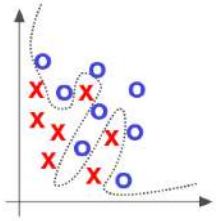
Over- and Under-fitting



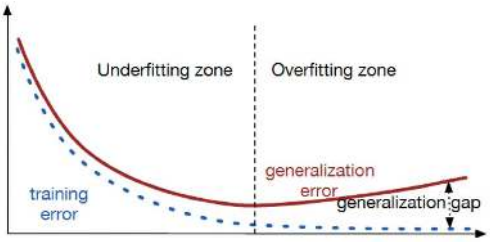
Underfitted



Appropriate



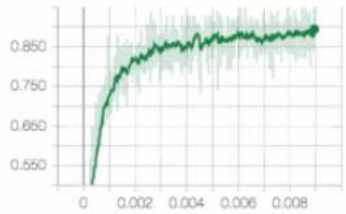
Overfitted



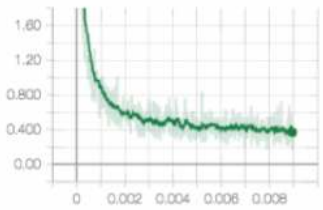
Learning Curves

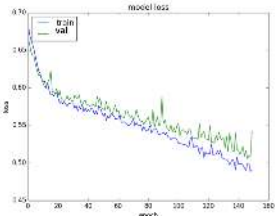
- Training graphs

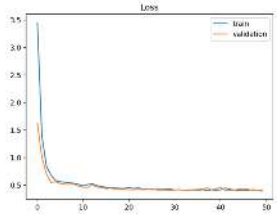
- Accuracy



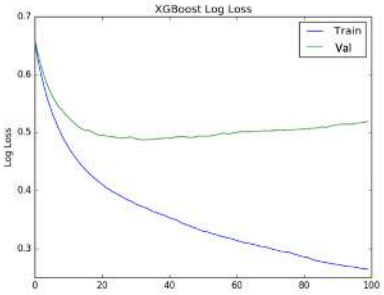
- Loss

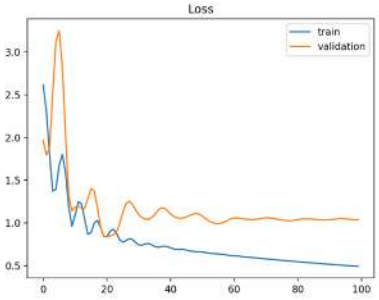




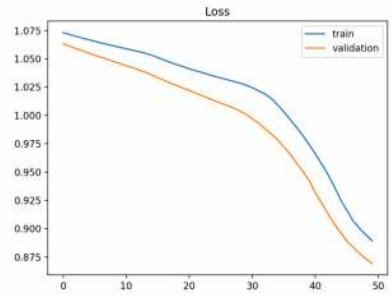


Overfitting Curves

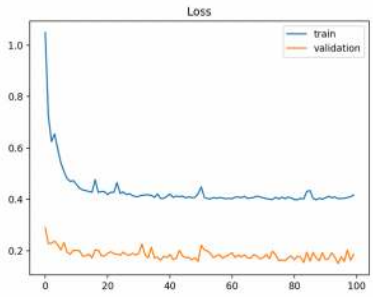




Other Curves



Underfitting (loss still decreasing)



Validation Set is easier than Training set

To Summarize

· Underfitting
- Training and validation losses decrease even at the end of the training

· Overfitting
- training loss decreases and validation loss increases

· Ideal training
- Small gap between training and validation loss, and both go down at same rate (stable without fluctuations).

· Bad signs
- Training error not going down
- Validation error not going down
- Performance on validation better than on training set
- Test on train set different than during training

. Bad practice
- Training set contain test data
- Debug algorithm on test data

Never touch during development or training

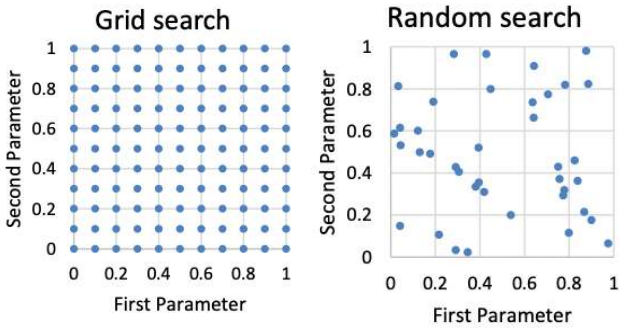
Hyperparameters

· Network architecture (e.g., num layers, #weights)
· Number of iterations
· Learning rate(s) (i.e., solver parameters, decay, etc.)
· Regularization
· Batch size
· …
· Overall: learning setup + optimization = hyperparameters.

Hyperparameters Tuning

· Methods:
- Manual search: most common
- Grid search (structured for 'real' applications)
  · Define ranges for all parameters spaces and select points.
  · Usually pseudo-uniformly distributed
  —> Iterate over all possible configurations

- Random search: Like grid search but one picks points at random in the predefined ranges.

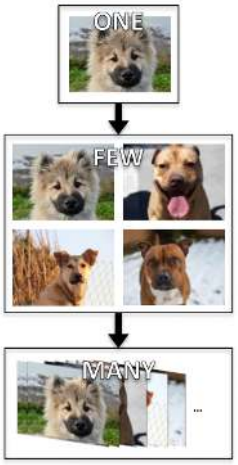


How to start

· Start with single training samples
- Check if output correct.
- Overfit —> train accuracy should be 100% because input just memorized.

· Increases to handful of samples (e.g., 4)
- Check if input is handled correctly

· Move from overfitting to more samples
- 5, 10, 100, 1000, …
- At some point, you should see generalization

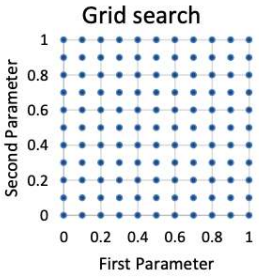


Find a Good Learning Rate

· Use all training data with small weight decay
· Perform initial loss sanity check e.g., log(C) for softmax with C classes
· Find a learning rate that makes the loss drop significantly (exponentially) within 100 iterations
· Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Coarse Grid Search

· Choose a few values of learning rate and weights decay around what worked from
· Train a few models for a few epochs.
· Good weight decay to try: 1e-4, 1e-5, 0



Refine Grid

· Pick best models found with coarse grid
· Refine grid search around these models
· Train them for longer (10-20 epochs)
Without learning rate decay
· Study loss curves <—most important debugging tool!

Timings

· How long does each iteration take?
    - Get precise timings!
    - If an iteration exceeds 500ms, things get dicey.

· Look for bottlenecks
    - Dataloadings: smaller resolution, compression, train from SSD
    - Backprop

· Estimate total time
    - How long until you see some pattern?
    - How long till convergence?

Network Architecture

· Frequent mistake: “Let’s use this super big network, train for two weeks and we see where we stand. “

· Instead: start with simplest network possible
    - Rule of thumb divide #layers you started with by 5

· Get debug cycles down
    - Ideally, minutes

Debugging

· Use train/validation/test curves
    - Evaluation needs to be consistent
    - Number need to be comparable

· Only make one change at a time
    - “I’ve added 5 more layers and double the training size, and now I also trained 5 days longer. Now it’s better, but why?”

· Visualize input, prediction, ground truth

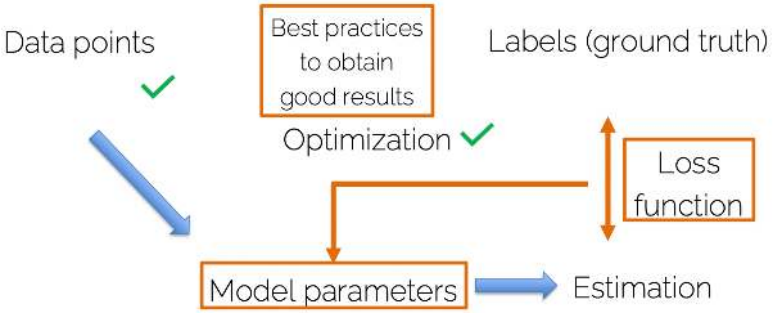
Common Mistakes in Practice

· Did not overfit to single batch first
· Forgot to toggle train/eval mode for network
    - Check later when we talk about dropout…
· Forgot to call .zero_grad() (in PyTorch) before calling .backward()
· Passed softmaxed outputs to a loss function that expects raw logits

LECTURE 7: Training Neural Networks II

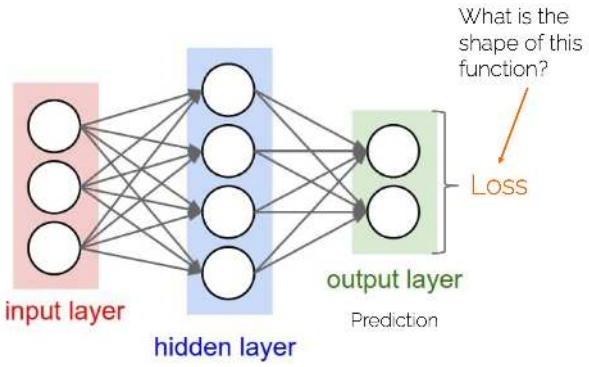
Training Neural Networks part 2

What we have seen so far



Output and Loss Functions

Neural Networks



Regression Losses

- L2 Loss: $L^2 = \sum_{i=1}^{n} \left(y_i - f(\boldsymbol{x}_i)\right)^2$
- L1 Loss: $L^1 = \sum_{i=1}^{n} |y_i - f(\boldsymbol{x}_i)|$

training pairs $[\boldsymbol{x}_i; y_i]$ (input and labels)

12	24	42	23
34	32	5	2
12	31	12	31
31	64	5	13

$f(\boldsymbol{x}_i)$

15	20	40	25
34	32	5	2
12	31	12	31
31	64	5	13

y_i

$$L^2(x, y) = 9 + 16 + 4 + 4 + 0 + \cdots + 0 = 33$$
$$L^1(x, y) = 3 + 4 + 2 + 2 + 0 + \cdots + 0 = 11$$

Regression Losses: L2 vs L1

- L2 Loss:

$$L^2 = \sum_{i=1}^{n} \left(y_i - f(\boldsymbol{x}_i)\right)^2$$

  - Sum of squared differences (SSD)
  - Prone to outliers
  - Compute-efficient optimization
  - Optimum is the mean

- L1 Loss:

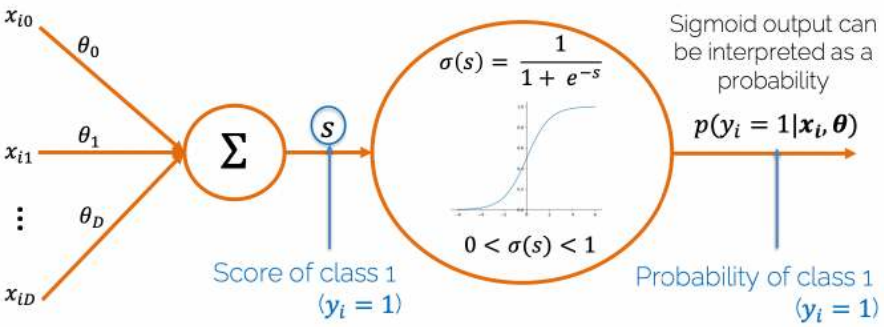
$$L^1 = \sum_{i=1}^{n} |y_i - f(\boldsymbol{x}_i)|$$

  - Sum of absolute differences
  - Robust (cost of outliers is linear)
  - Costly to optimize
  - Optimum is the median

Binary Classification: Sigmoid

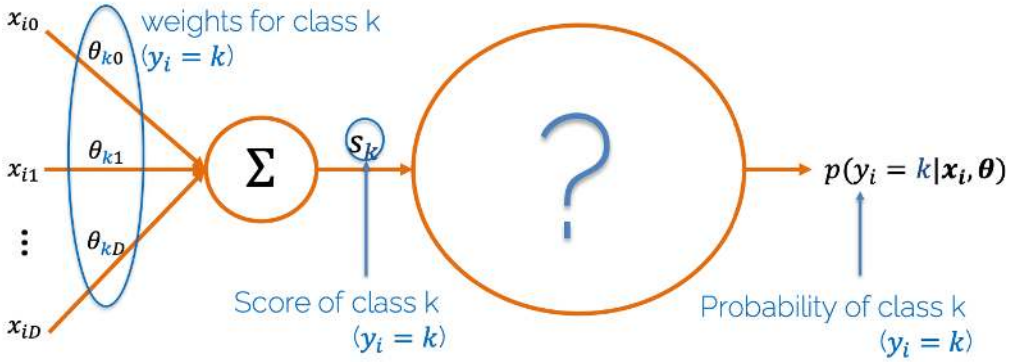
training pairs $[\boldsymbol{x}_i; y_i]$,
$\boldsymbol{x}_i \in \mathbb{R}^D, y_i \in \{1,0\}$ (2 classes)

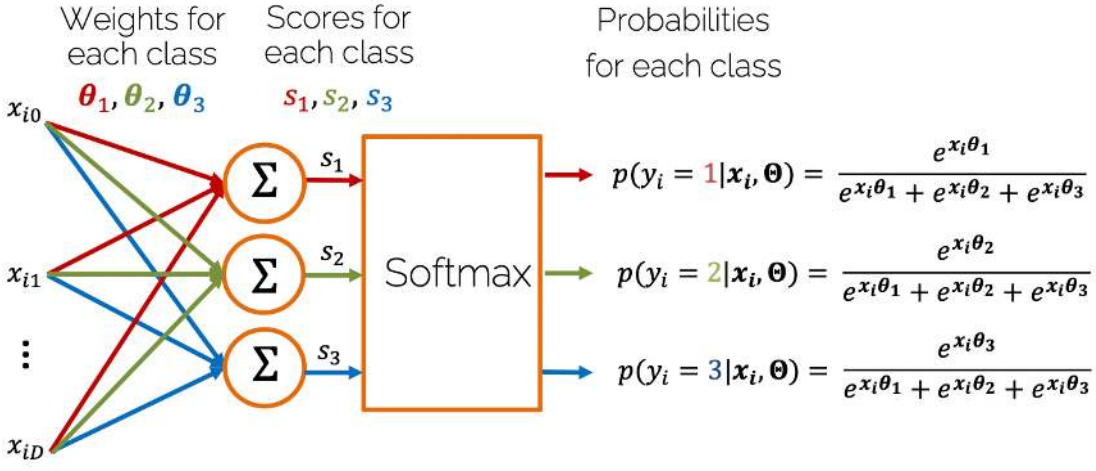
$$p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) = \sigma(s) = \frac{1}{1 + e^{-\sum_{d=0}^{D} \theta_d x_{id}}}$$



Multiclass Classification: Softmax

training pairs $[\boldsymbol{x}_i; y_i]$,
$\boldsymbol{x}_i \in \mathbb{R}^D, y_i \in \{1,2 \dots C\}$ (C classes)





- Softmax

$$p(y_i|\boldsymbol{x_i},\Theta)=\frac{e^{s_{y_i}}}{\sum_{k=1}^{C}e^{s_k}}=\frac{e^{\boldsymbol{x_i}\boldsymbol{\theta}_{y_i}}}{\sum_{k=1}^{C}e^{\boldsymbol{x_i}\boldsymbol{\theta}_k}}$$

Exp

normalize

Probability of the true class

training pairs $[\boldsymbol{x_i};y_i]$,
$\boldsymbol{x_i}\in\mathbb{R}^D, y_i\in\{1,2\dots C\}$
y_i: label (true class)

Parameters:
$\boldsymbol{\Theta}=[\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\dots,\boldsymbol{\theta}_C]$
$\boldsymbol{C}$: number of classes
$\boldsymbol{s}$: score of the class

1. Exponential operation: make sure probability>0
2. Normalization: make sure probabilities sum up to 1.

- Numerical Stability

$$p(y_i|\boldsymbol{x_i},\Theta)=\frac{e^{s_{y_i}}}{\sum_{k=1}^{C}e^{s_k}}=\frac{e^{s_{y_i}-s_{max}}}{\sum_{k=1}^{C}e^{s_k-s_{max}}}$$

Try to prove it by yourself ☺

- Cross-Entropy Loss (Maximum Likelihood Estimate)

$$L_i=-\log(p(y_i|\boldsymbol{x_i},\Theta))=-\log(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}})$$

Hinge Loss (SVM LOSS)

· Score function $\boldsymbol{s}=\boldsymbol{f}(\boldsymbol{xi}\ ,\ \boldsymbol{\Theta})$
– e.g., $\boldsymbol{f}\ \boldsymbol{xi},\boldsymbol{\Theta}=[xi0,xi1,\dots,xid]\cdot[\theta\mathbf{1},\theta\mathbf{2},\dots,\theta\boldsymbol{C}]$
· Hinge Loss (Multiclass SVM Loss)

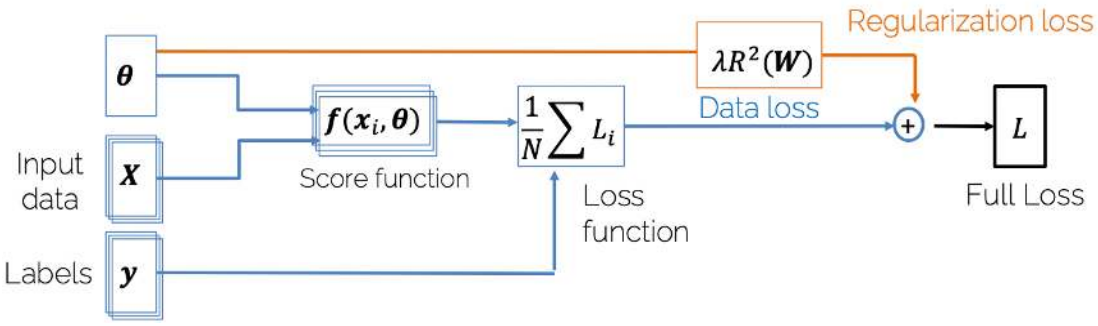
$$L_i=\sum_{k\neq y_i}\max(0,s_k-s_{y_i}+1)$$

Multiclass Classification: Hinge vs Cross-Entropy

- Hinge Loss: $L_i=\sum_{k\neq y_i}\max(0,s_k-s_{y_i}+1)$
- Cross Entropy Loss: $L_i=-\log(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}})$

Cross Entropy *always* wants to improve! (Loss never 0), Hinge Loss saturates.

Loss in Compute Graph

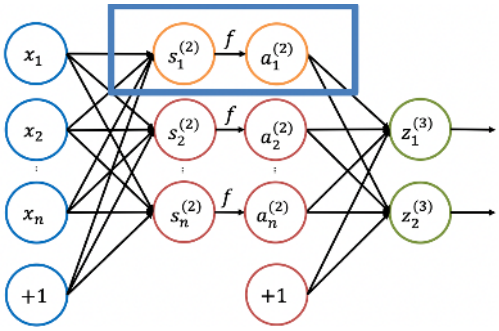


Want to find optimal θ (weights are unknown of optimization problem)
- Compute gradient w.r.t θ.
- Gradient $\nabla\theta L$ is computed via backpropagation

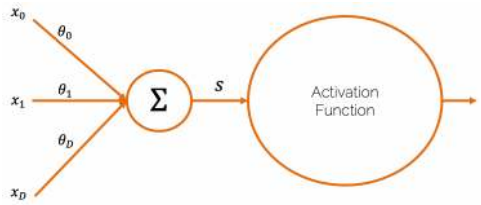
- Score function $\boldsymbol{s = f(x_i, \theta)}$ Given a function with weights $\boldsymbol{\theta}$, Training pairs $[\boldsymbol{x_i}; \boldsymbol{y_i}]$ (input and labels)
- Data Loss - Cross Entropy $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}})$
  - SVM $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$
- Regularization Loss: e.g., $L2$-Reg: $R^2(W) = \sum w_i^2$
- Full Loss $L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R^2(W)$
- Full Loss = Data Loss + Reg Loss

Activation Functions

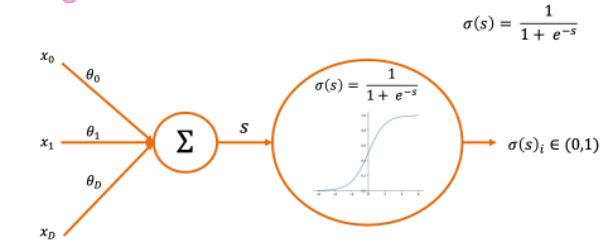
Neural Networks

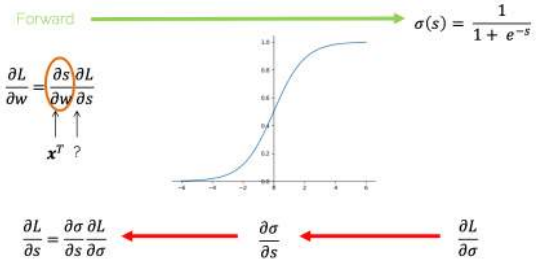


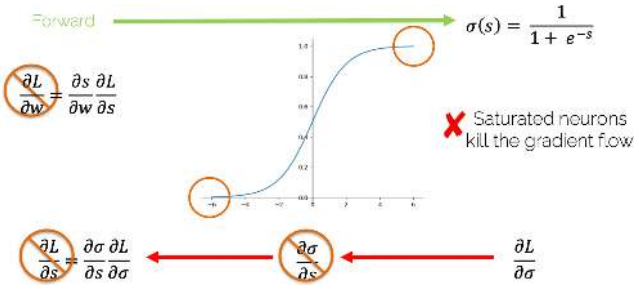
Activation Functions or Hidden Units

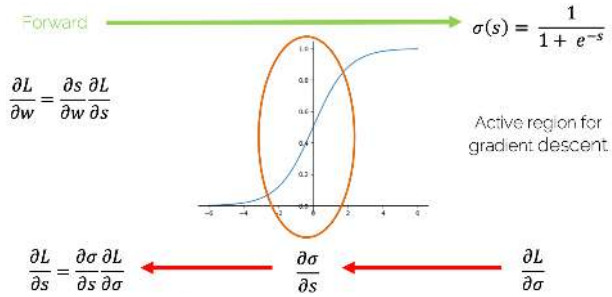


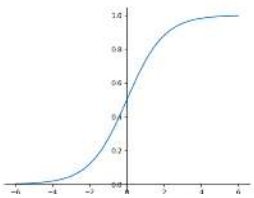
Sigmoid Activation









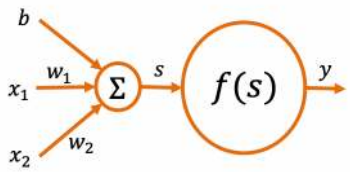


$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

Output is always positive!

- Sigmoid output provides positive input for the next layer

Sigmoid Output not Zero-centered

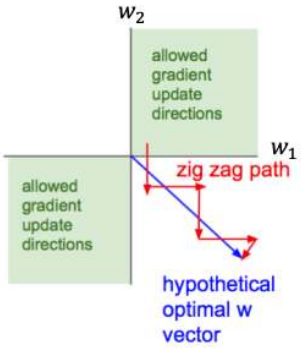


Assume we have all positive data:
$\boldsymbol{x} = (x_1, x_2)^T > 0$

either positive or negative

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_1} \quad x_1 > 0$$

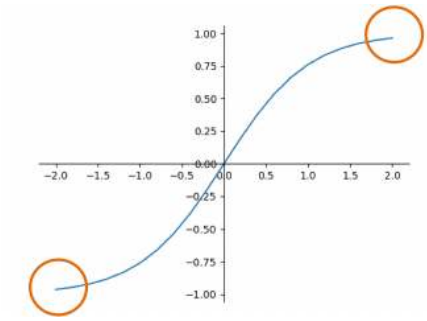
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_2} \quad x_2 > 0$$



w_1, w_2 can only be increased or decreased at the same time, which is not good for update.

That is also why you need zero-centered data.

Tanh Activation

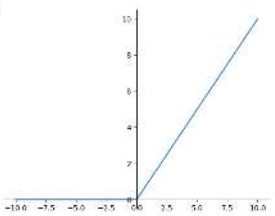


✘ Still saturates

✓Zero-centered

Rectified Linear Units (ReLU)

$$\sigma(x) = \max(0, x)$$



Large and consistent gradients ✓

✓ Fast convergence ✓ Does not saturate

✘ Dead ReLU

What happens if a ReLU outputs zero?

Large and consistent gradients ✓

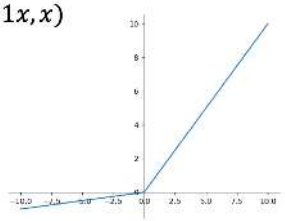
✓ Fast convergence ✓ Does not saturate

- Initializing ReLU neurons with slightly positive biases (0.01) makes it likely that they stay active for most inputs

$$f\left(\sum_i w_i x_i + b\right)$$

Leaky ReLU

$$\sigma(x) = \max(0.01x, x)$$

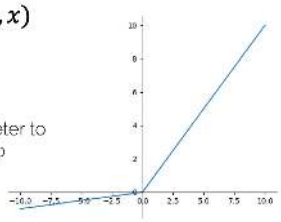


✓ Does not die

Parametric ReLU

$$\sigma(x) = \max(\alpha x, x)$$

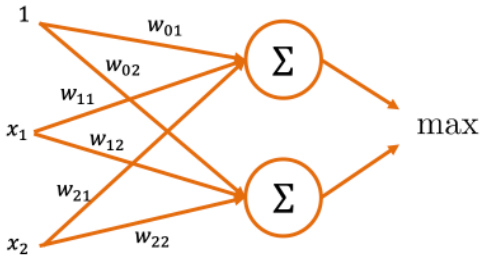
One more parameter to backprop into

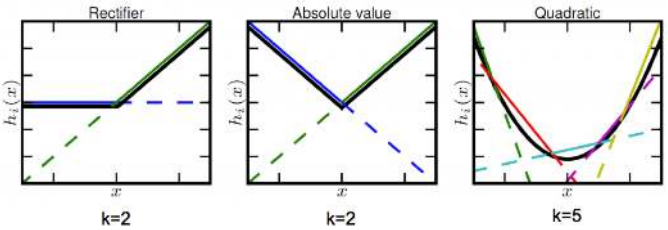


✓ Does not die

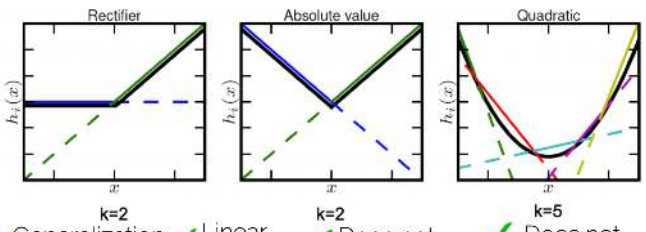
Maxout Units

$$Maxout = \max(w_1^T x + b_1, w_2^T x + b_2)$$





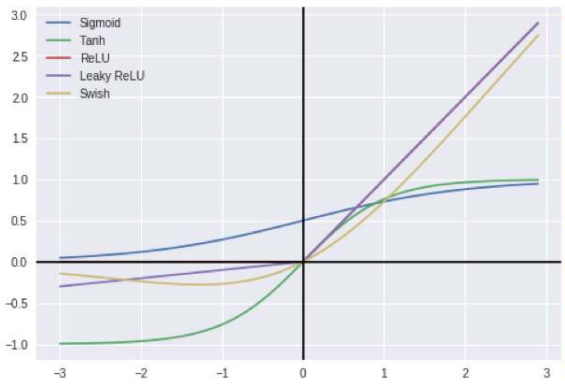
Piecewise linear approximation of a convex function with N pieces



✓ Generalization of ReLUs ✓ Linear regimes ✓ Does not die ✓ Does not saturate

✘ Increases of the number of parameters

In a Nutshell



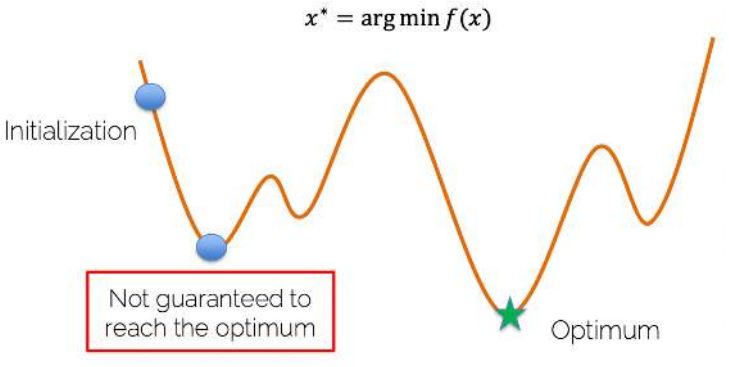
ACTIVATION FUNCTION	EQUATION	RANGE
Linear Function	$f(x) = x$	$(-\infty, \infty)$
Step Function	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$	$\{0, 1\}$
Sigmoid Function	$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$
Hyperbolic Tanjant Function	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$(-1, 1)$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU	$f(x) = \begin{cases} 0.01 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Swish Function	$f(x) = 2x\sigma(\beta x) = \begin{cases} \beta = 0 \text{ for } f(x) = x \\ \beta \to \infty \text{ for } f(x) = 2\max(0, x) \end{cases}$	$(-\infty, \infty)$

Quick Guide

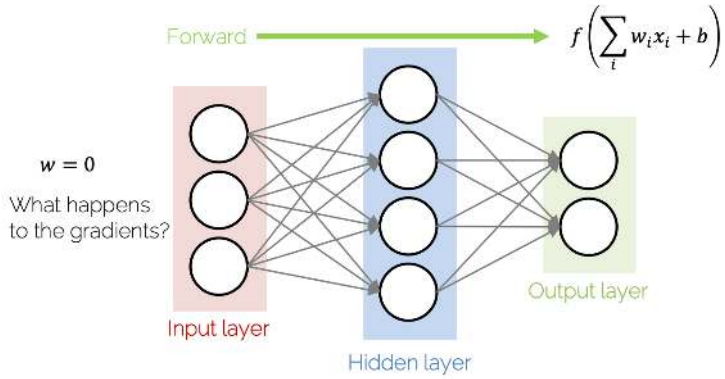
· Sigmoid/TanH are not really used in feedforward nets.
· ReLU is the standard choice.
· Second choice are the variants of ReLU or Maxout.
· Recurrent nets will require Sigmoid/TanH or similar.

Weights Initialization

Initialization is Extremely Important!



How do I start?



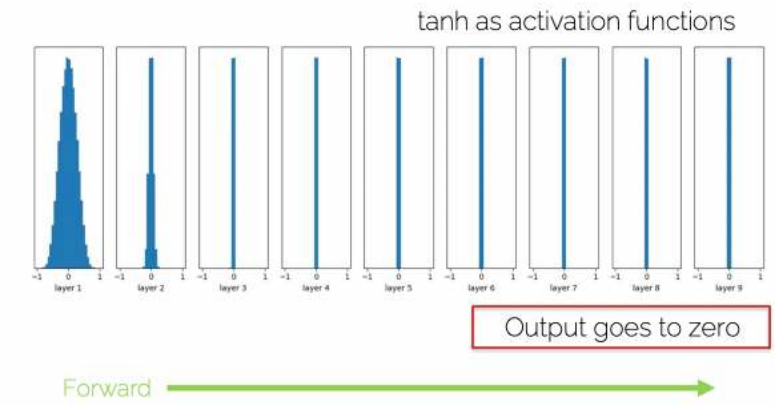
All Weights Zero

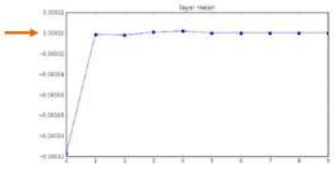
·What happens to the gradients?
· The hidden units are all going to compute the same function, gradients are going to be the same
    - No symmetry breaking

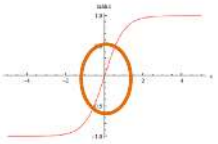
Small Random Numbers

· Gaussian with zero mean and standard deviation 0.01

· Let's see what happens:
    - Network with 10 layers with 500 neurons each
    - Tanh as activation functions
    - Input unit Gaussian data





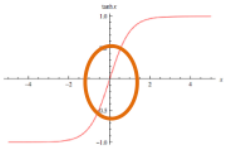


Small w_i^l cause small output for layer l.

$$f_l\left(\sum_i w_i^l x_i^l + b^l\right) \approx 0$$

Forward

Even activation function's gradient is ok, we still have vanishing gradient problem.



Small outputs of layer l (input of layer $l+1$) cause small gradient w.r.t to the weights of layer $l+1$:

$$f_{l+1}\left(\sum_i w_i^{l+1} x_i^{l+1} + b^{l+1}\right)$$

$$\frac{\partial L}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot \frac{\partial f_{l+1}}{\partial w_i^{l+1}} = \frac{\partial L}{\partial f_{l+1}} \cdot x_i^{l+1} \approx 0$$

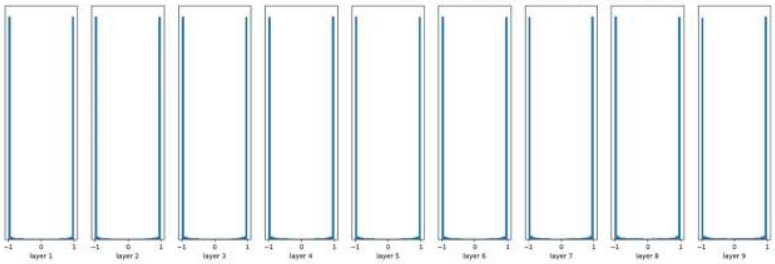
Vanishing gradient, caused by small output

Backward

Big Random Numbers

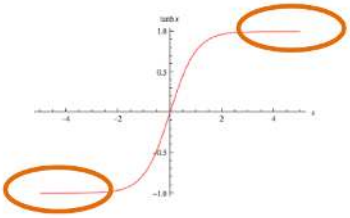
· Gaussian with zero mean and standard deviation 1
· Let us see what happens:
- Network with 10 layers with 500 neurons each
- Tanh as activation functions
- Input unit Gaussian data

tanh as activation functions



Output saturated to -1 and 1

Output saturated to -1 and 1. Gradient of the activation function becomes close to 0.



$$f(s) = f\left(\sum_i w_i x_i + b\right)$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial s} \cdot \frac{\partial s}{\partial w_i} \approx 0$$

Vanishing gradient, caused by saturated activation function.

Xavier Initialization

- Gaussian with zero mean, but what standard deviation?

$$Var(s) = Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i)$$

Notice: n is the number of input neurons for the layer of weights you want to initialized. This n is not the number N of input data $X \in R^{N \times D}$. For the first layer $n = D$.

Tips:

$$E[X^2] = Var[X] + E[X]^2$$

If X, Y are independent:

$$Var[XY] = E[X^2 Y^2] - E[XY]^2$$
$$E[XY] = E[X]E[Y]$$

- Gaussian with zero mean, but what standard deviation?

$$Var(s) = Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i)$$
$$= \sum_i^n [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i)$$

Zero mean Zero mean

$$Var(s) = Var\left(\sum_i^n w_i x_i\right) = \sum_i^n Var(w_i x_i)$$
$$= \sum_i^n [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i)$$
$$= \sum_i^n Var(x_i)Var(w_i) = n(Var(w)Var(x))$$

Identically distributed (each random variable has the same distribution)

(Glorot and Bengio, AISTATS'10) Xavier Initialization

- How to ensure the variance of the output is the same as the input?

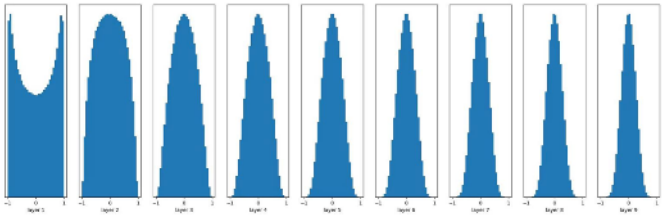
Goal:

$Var(s) = Var(x)$ ⟶ $\underbrace{n \cdot Var(w)}_{=1} Var(x) = Var(x)$

⟶ $Var(w) = \frac{1}{n}$

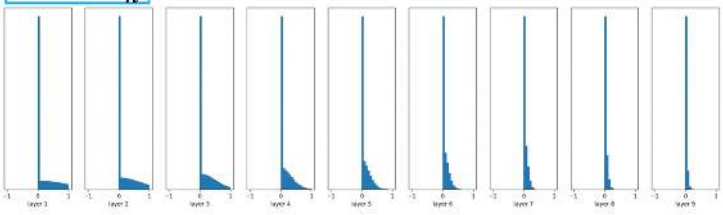
n: number of input neurons

$Var(w) = \frac{1}{n}$ tanh as activation functions



Xavier Initialization with ReLU (Kaiming Initialization)

$Var(w) = \frac{1}{n}$

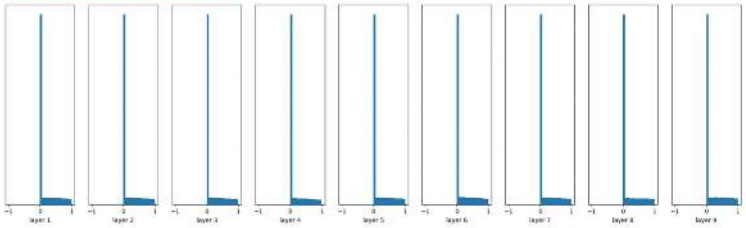


ReLU kills Half of the Data
What's the solution?

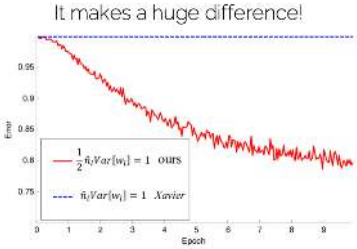
When using ReLU, output close to zero again ☹

Kaiming Initialization with ReLU

$Var(w) = \frac{1}{n/2} = \frac{2}{n}$



$Var(w) = \frac{2}{n}$



- Use ReLU and Xavier/2 initialization

Summary

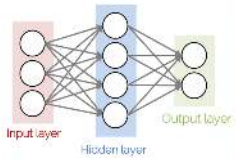


Image Classification	Output Layer	Loss function
Binary Classification	Sigmoid	Binary Cross entropy
Multiclass Classification	Softmax	Cross entropy

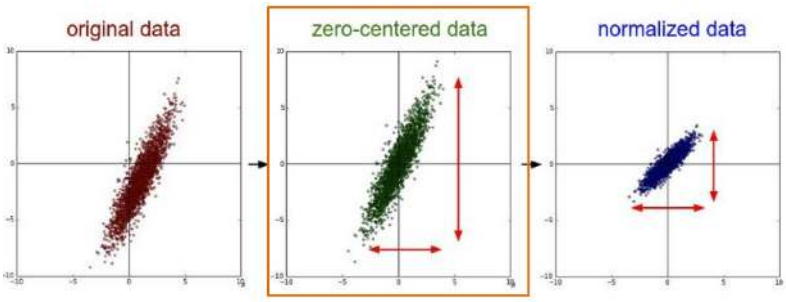
Other Losses:
SVM Loss (Hinge Loss), L1/L2-Loss

Initialization of optimization
- How to set weights at beginning

LECTURE 8: Training Neural Networks III

Data Augmentation

Data Pre-Processing

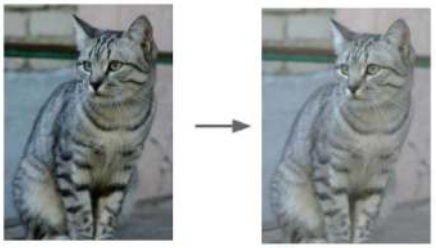


Data Augmentation

· A classifier has to be invariant to a wide variety of transformations
· Helping the classifier: synthesize data simulating plausible transformations.

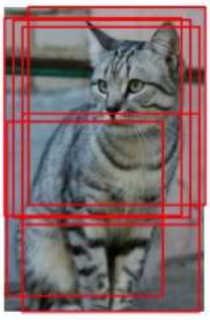
Data Augmentation: Brightness

· Random brightness and contrast changes



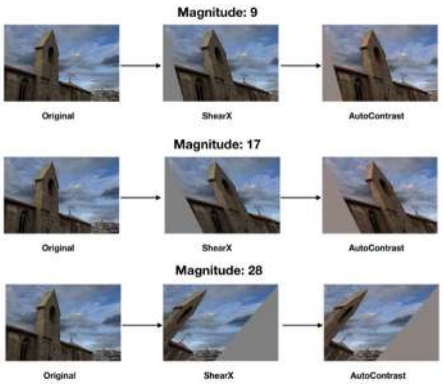
Data Augmentation: Random Crops

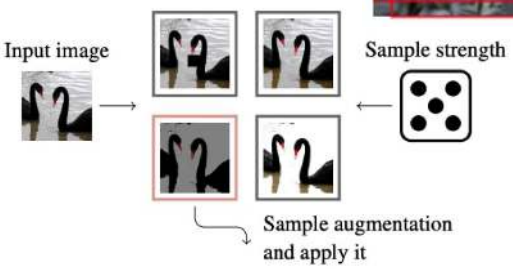
· Training: random crops
- Pick a random L in [256,480]
- Resize training image, short side L
- Randomly sample crops of 224x224



· Testing: fixed set of crops
- Resize image at N scale
_ 10 fixes crops of 224x224 (4 corners + 1 center) x 2 flips

Data Augmentation: Advanced





Data Augmentation

· When comparing two networks make sure to use the same data augmentation

· Consider data augmentation a part of your network design

Advanced Regularization

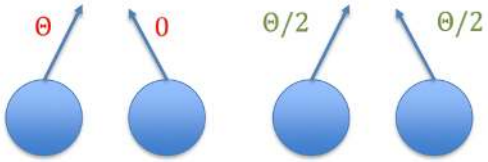
L2 regularization, also (wrongly) called weight decay

· L2 regularization

$$\Theta_{k+1} = \Theta_k - \epsilon \nabla_\Theta (\Theta_k, x, y) - \lambda \theta_k$$

Learning rate | Gradient | Gradient of L2-regularization

· Penalize large weights
· Improves generalization



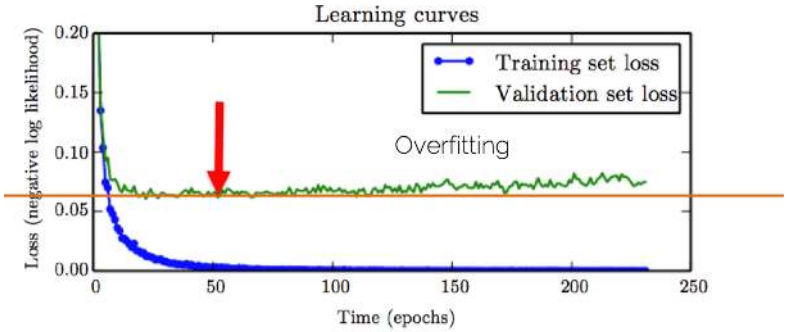
· Weight decay regularization

$$\Theta_{k+1} = (1 - \lambda)\Theta_k - \alpha \nabla f_k(\Theta_k)$$

Learning rate of weight decay | Learning rate of the optimizer

· Equivalent to L2 regularization in GD, but not in Adam

Early Stopping



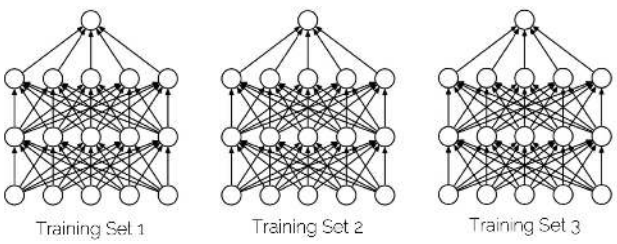
Bagging and Ensemble Methods

· Train multiple models and average their results.

· E.g., use a different algorithm for optimization or change the objective function / loss function.

· If errors are uncorrelated, the expected combined error will decrease linearly with the ensemble size.

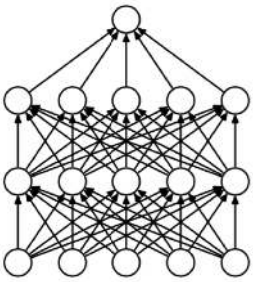
· Bagging: uses k different datasets (or SGD/init noise)



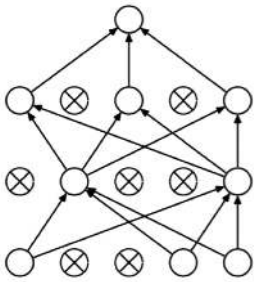
Dropout

Dropout

· Disable a random set of neurons (typically 50%)



(a) Standard Neural Net



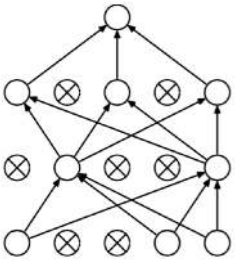
(b) After applying dropout.

Dropout: Intuition

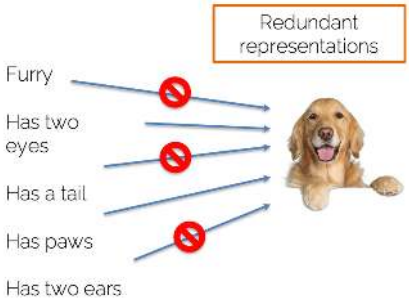
· Using half the network = half capacity
- Redundant representations
- Base your scores on more features

· Consider it as a model ensemble
- Training a large ensemble of models, each on different set of data (mini-batch) and with SHARED parameters.

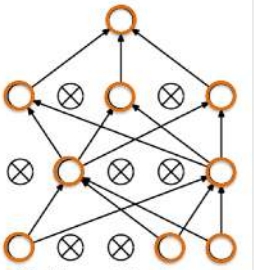
Reducing co-adaptation between neurons



(b) After applying dropout.



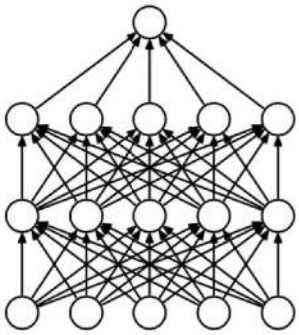
· two models in one



(b) After applying dropout.

Dropout: Test Time

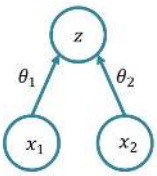
· All neurons are “turned on” - no dropout



Conditions at train and test time are not the same

- Test:
- Train:

PyTorch: model.train() and model.eval()



Weight scaling inference rule

$$z = (\theta_1 x_1 + \theta_2 x_2) \cdot p$$

Dropout probability $p = 0.5$

$$E[z] = \frac{1}{4}(\theta_1 0 + \theta_2 0 + \theta_1 x_1 + \theta_2 0 + \theta_1 0 + \theta_2 x_2 + \theta_1 x_1 + \theta_2 x_2) = \frac{1}{2}(\theta_1 x_1 + \theta_2 x_2)$$

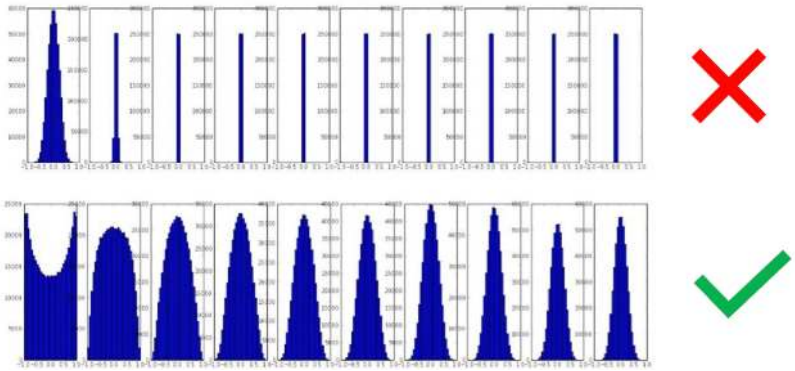
Dropout: Before
· Efficient bagging method with parameters sharing
· Try it!
· Dropout reduces the effective capacity of a model, but needs more training time
· Efficient regularization method, can be used with L2

Dropout: Nowadays
· Usually does not work well when combined with batch-norm.
· Training takes a bit longer, usually 1.5x
· But can be used for uncertainty estimation.
· Monte Carlo dropout (Yarin Gal and Zoubin Ghahramani series of papers)

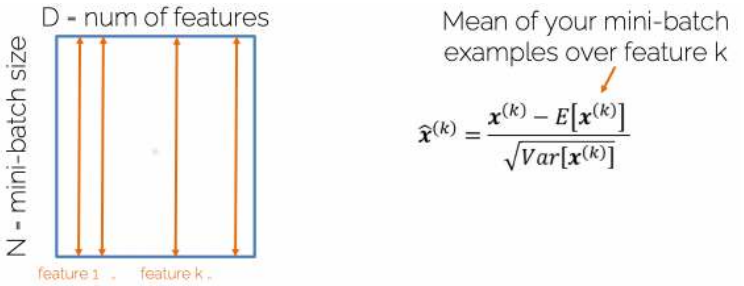
Monte Carlo Dropout
· Neural networks are massively overconfident
· We can use dropout to make the softmax probabilities more calibrated
· Training: use dropout with a low p (0.1 or 0.2).
· Inference, run the same image multiple times (25-100), and average the results.

Batch Normalization. Reducing Internal Covariate Shift

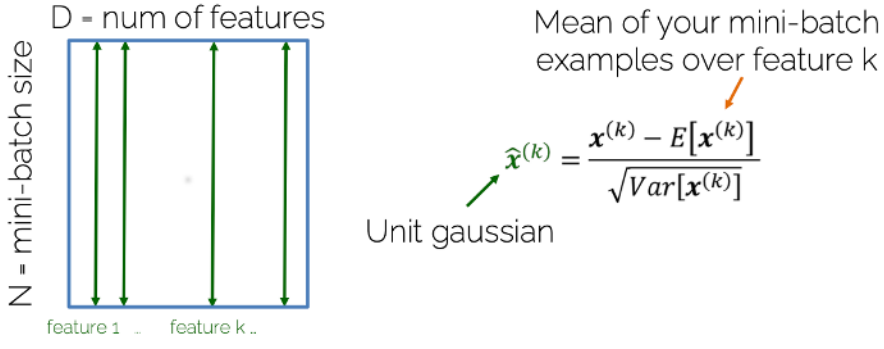
Our Goal
· All we want is that our activations do not die out



Batch Normalization
· Wish: Unit Gaussian activations (in our example).
· Solution: let's do it



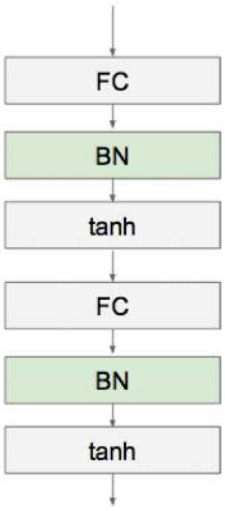
· In each dimension of the features, you have a unit gaussian (in our example)



· I each dimension of the features, you have a unit gaussian (in our example)
· For NN in general, BN normalizes the mean and variance of the inputs to your activation functions

BN Layer

· A layer to be applied after Fully Connected (or Convolutional) layers and before non-linear activation functions



Batch Normalization

- 1. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

← Differentiable function so we can backprop through it...

- 2. Allow the network to change the range

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

← These parameters will be optimized during backprop

The network *can* learn to undo the normalization

$$\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$$

$$\beta^{(k)} = E[x^{(k)}]$$

· OK to treat dimensions separately?
Shown empirically that even if features are not correlated, convergence is still faster with this method.

BN: Train vs Test

· Train time: mean and variance is taken over the mini-batch $\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$

· Test-time: what happens if we can just process one image at a time?
- No chance to compute a meaningful mean and variance.

Training: Compute mean and variance from mini-batch 1,2,3 …

Testing: Compute mean and variance by running an exponentially weighted averaged across training mini-batches. Use them as σ^2_{test} and μ_{test}

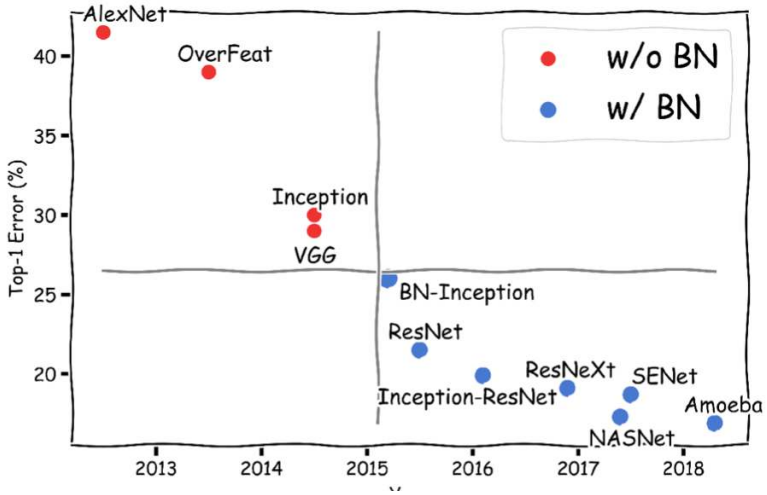
$$Var_{running} = \beta_m * Var_{running} + (1 - \beta_m) * Var_{minibatch}$$
$$\mu_{running} = \beta_m * \mu_{running} + (1 - \beta_m) * \mu_{minibatch}$$

β_m: momentum (hyperparameter)

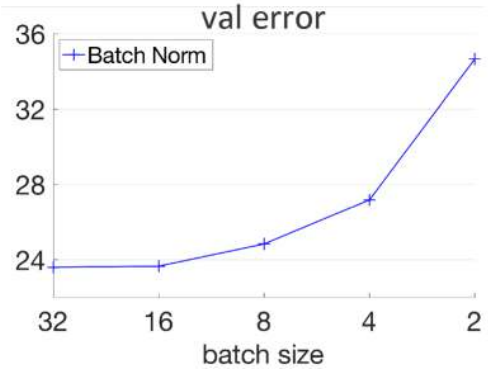
BN: What do you get?

· Very deep nets are much easier to train, more stable gradients
· A much larger range of hyperparameters works similarly when using BN

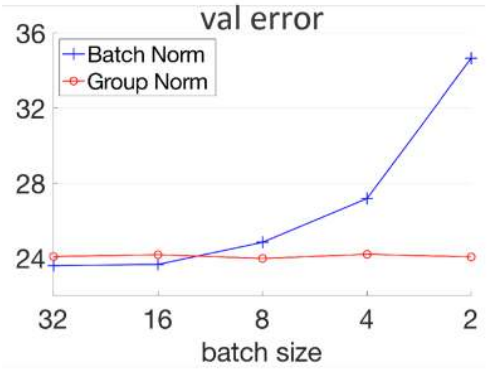
BN: A Milestone

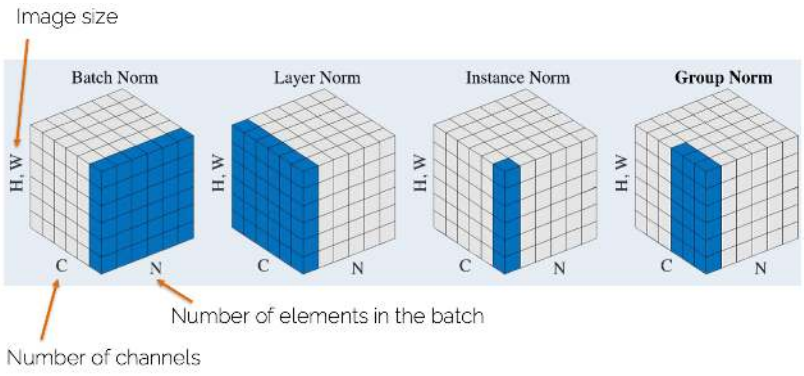


BN: Drawbacks



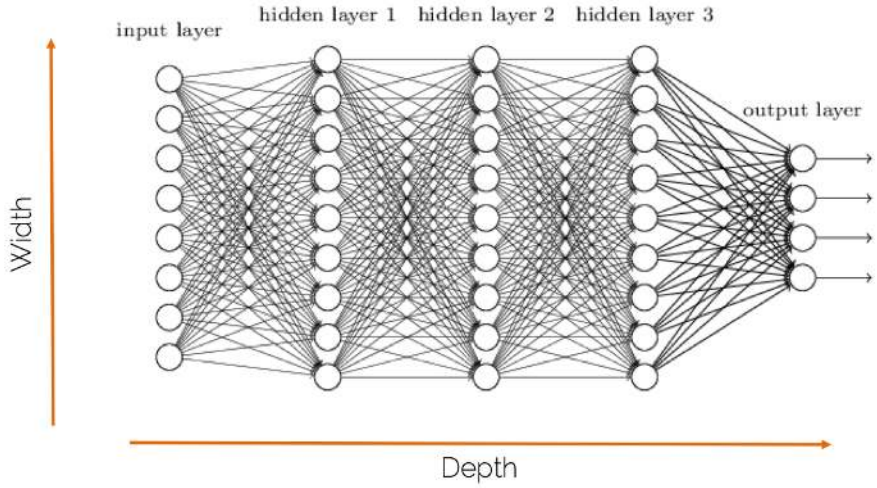
Other Normalizations



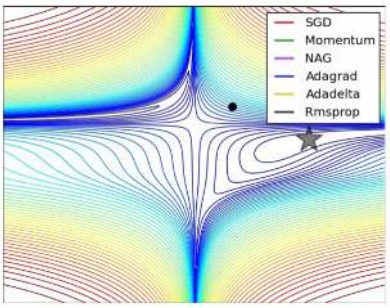


What We Know

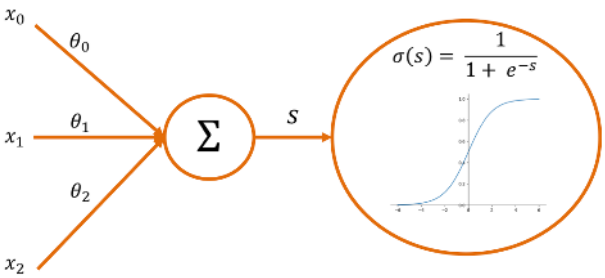
What do we know so far?



SGD Variations (Momentum, etc.)



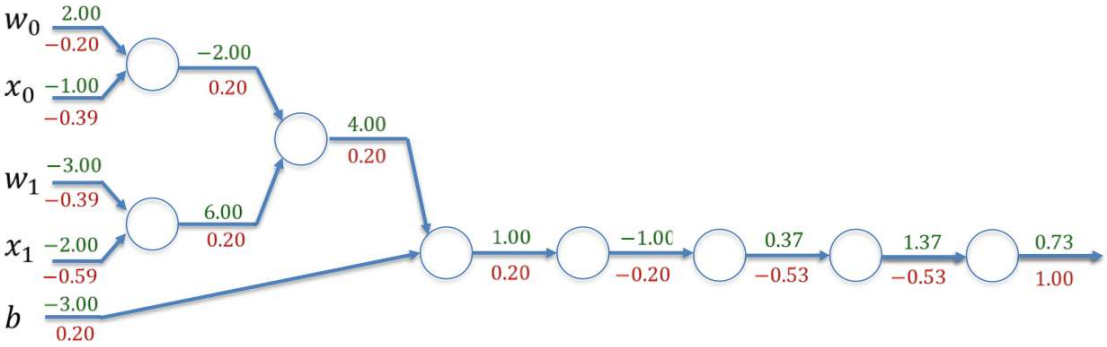
Concept of a 'Neuron'



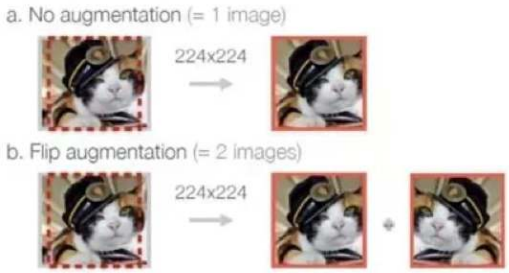
Activation Functions (non-linearities)

- Sigmoid: $\sigma(x) = \frac{1}{(1+e^{-x})}$
- ReLU: $\max(0, x)$
- TanH: $\tanh(x)$
- Leaky ReLU: $\max(0.1x, x)$

Backpropagation



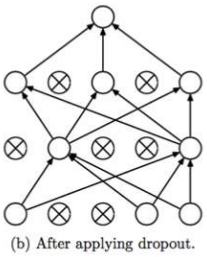
Data Augmentation



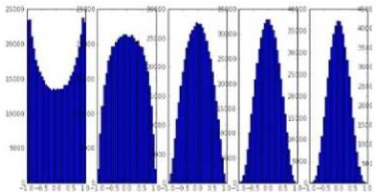
Batch-Norm

$$\hat{\boldsymbol{x}}^{(k)} = \frac{\boldsymbol{x}^{(k)} - E[\boldsymbol{x}^{(k)}]}{\sqrt{Var[\boldsymbol{x}^{(k)}]}}$$

Dropout



Weight Initialization (e.g., Kaiming)



Weight Regularization

e.g., L^2-reg: $R^2(\boldsymbol{W}) = \sum_{i=1}^{N} w_i^2$

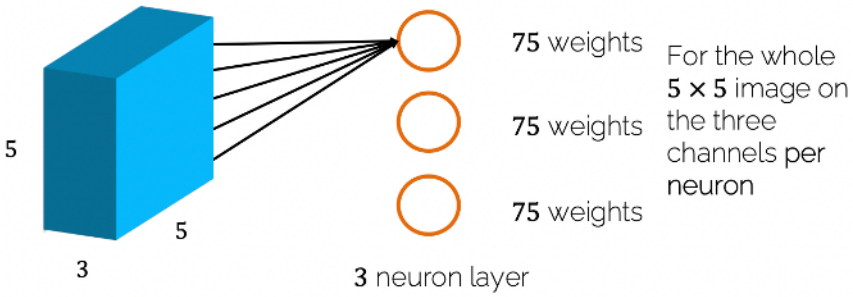
Why not simply more layers?

· Neural nets with at least one hidden layer are universal function approximators.
· But generalization is another issue.
· Why not just go deeper and get better?
  - No structure!!
  - it is just brute force!
  - optimization becomes hard
  - performance plateaus/ drops!
· We need more! More means CNNs, RNNs and eventually Transformers.

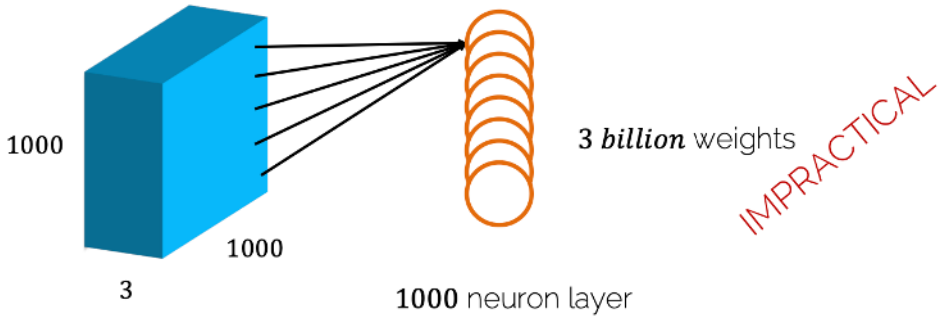
LECTURE 9: Convolutional Neural Networks

Problems using FC Layers on Images

· How to process a tiny image with FC layers



· How to process a normal image with FC layers



Why not simply more FC Layers ?

We cannot make networks arbitrarily complex

· Why not just go deeper and get better?

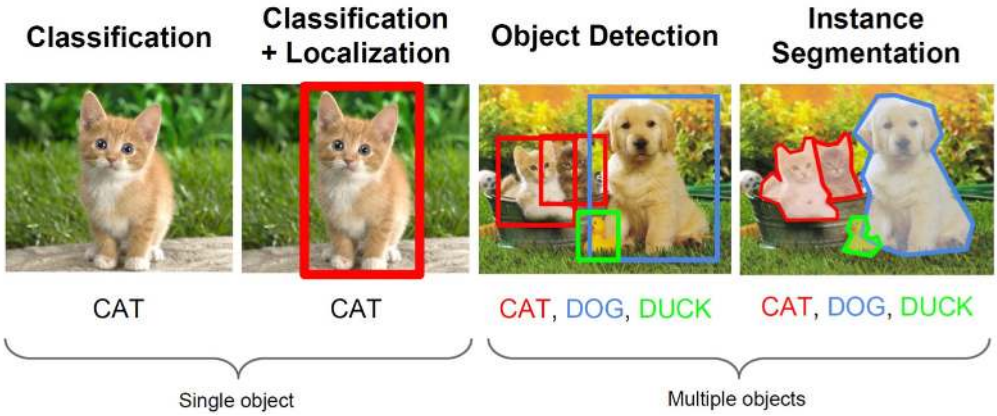
- No structure !!
- It is just brute force!
- Optimization becomes hard
- Performance plateaus/drops!

Better Way than FC?

· We want to restrict the degrees of freedom

- We want a layer with structure
- Weight sharing —> using the same weights for different parts of the image

Using CNNs in Computer Vision

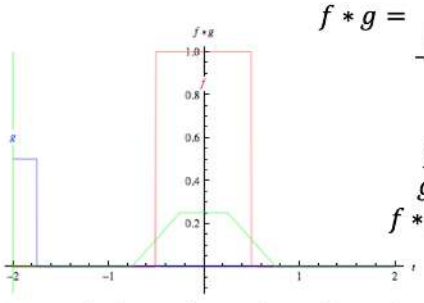


Convolutions

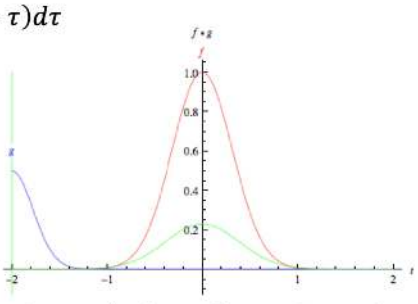
What are Convolutions?

$$f * g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

f = red
g = blue
$f * g$ = green



Convolution of two box functions



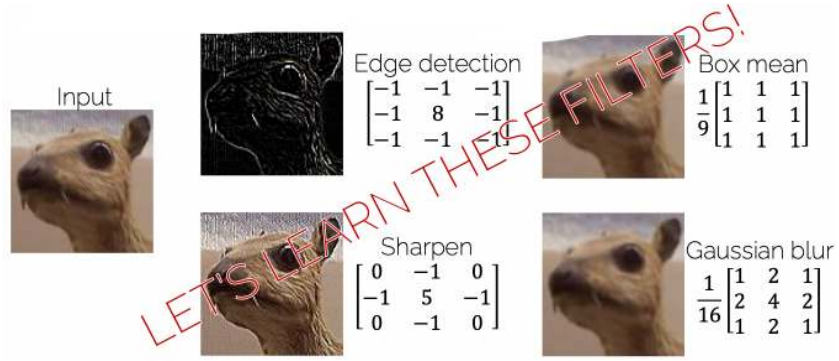
Convolution of two Gaussians

Application of a filter to a function
— The 'smaller' one is typically called the filter kernel

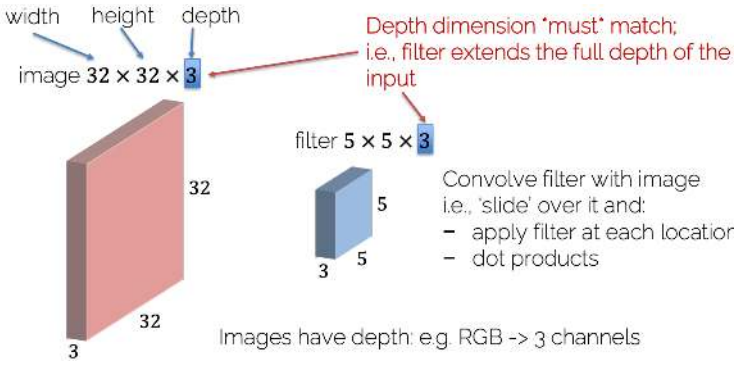
LOOK FOR THE EXAMPLE ON PAGE 14-25 also LOOK FOR THE EXAMPLE OF CONVOLUTION ON IMAGES ON PAGE 26-34

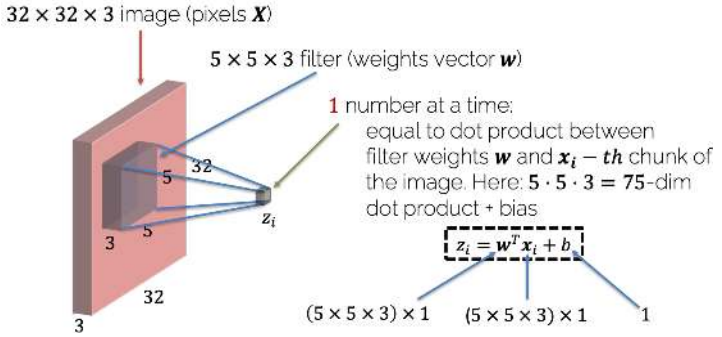
Image Filters

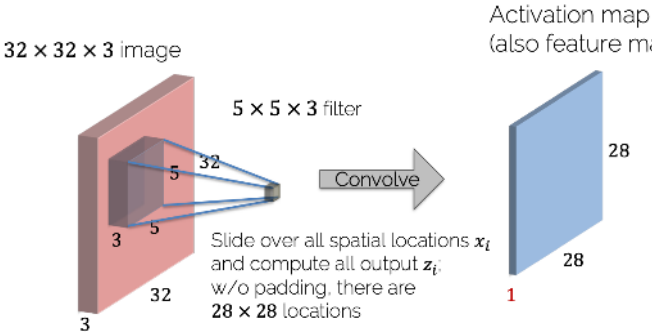
· Each kernel gives us a different image filter



Convolutions on RGB images

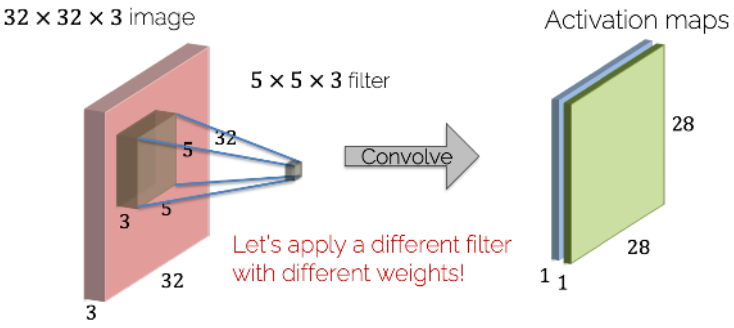


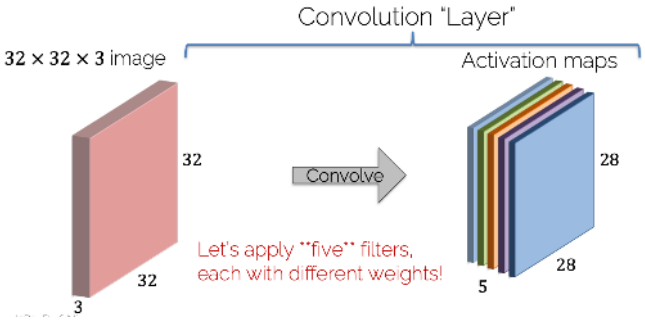




Convolution Layer

Convolution Layer





· A basic layer is defined by
  - Filter width and height (depth is implicitly given)
  - Number of different filter banks (#weight sets)

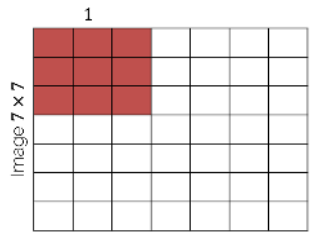
· Each filter captures a different image characteristic

Different Filters

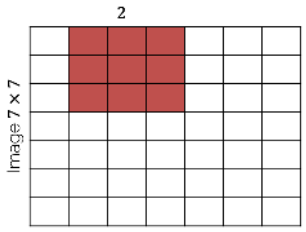
· Each filter captures different image characteristics:
- Horizontal edges
- Vertical edges
- Circles
- Squares
- …

Dimensions of a Convolution Layer

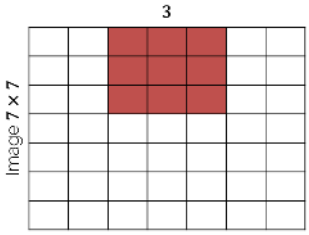
Convolutional Layer: Dimensions



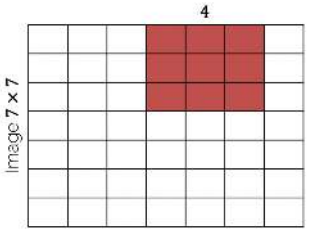
Input: 7 × 7
Filter: 3 × 3
Output: 5 × 5



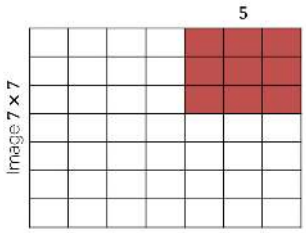
Input: 7 × 7
Filter: 3 × 3
Output: 5 × 5



Input: 7 × 7
Filter: 3 × 3
Output: 5 × 5

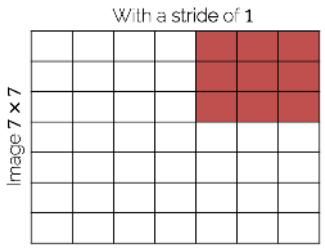


Input: 7 × 7
Filter: 3 × 3
Output: 5 × 5



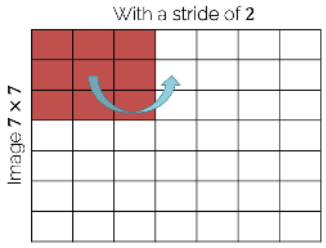
Input: 7 × 7
Filter: 3 × 3
Output: 5 × 5

Convolution Layer: Strides

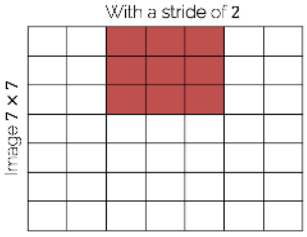


Input: 7 × 7
Filter: 3 × 3
Stride: 1
Output: 5 × 5

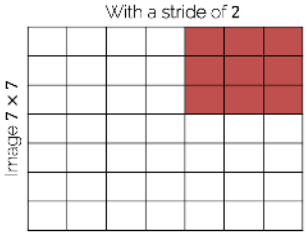
Stride of S: apply filter every S-th spatial location; i.e. subsamp.e the image



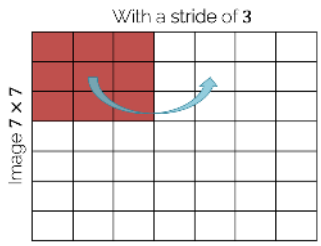
Input: 7 × 7
Filter: 3 × 3
Stride: 2
Output: 3 × 3



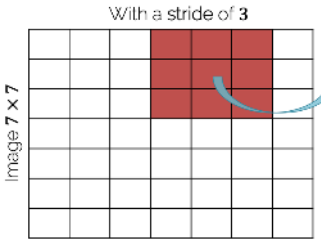
Input: 7 × 7
Filter: 3 × 3
Stride: 2
Output: 3 × 3



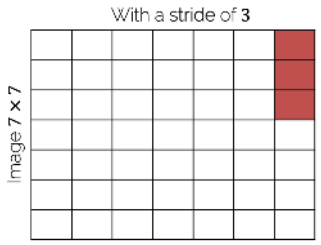
Input: 7 × 7
Filter: 3 × 3
Stride: 2
Output: 3 × 3



Input: 7 × 7
Filter: 3 × 3
Stride: 3
Output: ? × ?



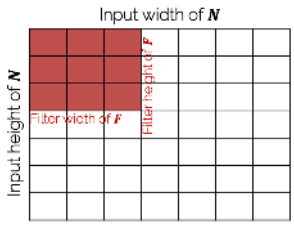
Input: 7 × 7
Filter: 3 × 3
Stride: 3
Output: ? × ?



Input: 7 × 7
Filter: 3 × 3
Stride: 3
Output: ? × ?

Does not really fit (remainder left)
→ Illegal stride for input & filter size!

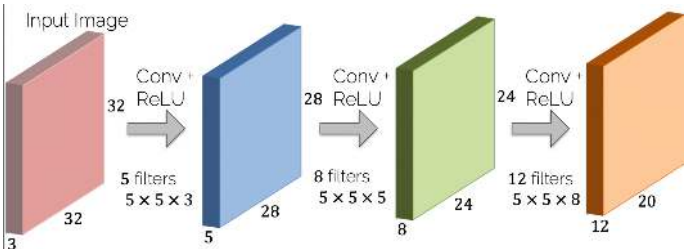
Convolution Layer: Dimensions



Input: $N \times N$
Filter: $F \times F$
Stride: S
Output: $\left(\frac{N-F}{S}+1\right) \times \left(\frac{N-F}{S}+1\right)$

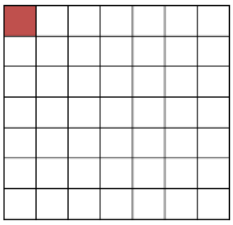
$N = 7, F = 3, S = 1$: $\frac{7-3}{1}+1 = 5$
$N = 7, F = 3, S = 2$: $\frac{7-3}{2}+1 = 3$
$N = 7, F = 3, S = 3$: $\frac{7-3}{3}+1 = 2.\overline{3}$

Fractions are illegal



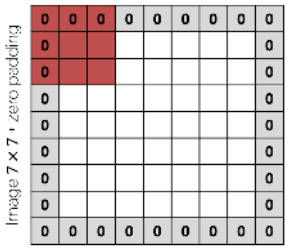
Shrinking down so quickly (32→28→24→20) is typically not a good idea.

Convolution Layers: Padding



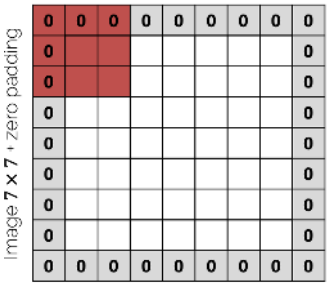
Why padding?

- Sizes get small too quickly
- Corner pixel is only used once



Why padding?

- Sizes get small too quickly
- Corner pixel is only used once



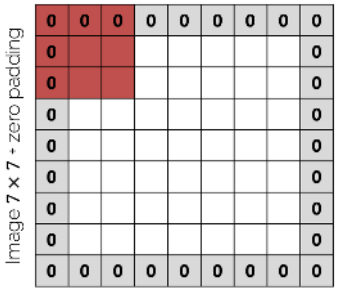
Input ($N \times N$): 7 × 7
Filter ($F \times F$): 3 × 3
Padding (P): 1
Stride (S): 1
Output 7 × 7

Most common is '*zero*' padding

Output Size:

$$\left(\left\lfloor\frac{N+2\cdot P-F}{S}\right\rfloor+1\right) \times \left(\left\lfloor\frac{N+2\cdot P-F}{S}\right\rfloor+1\right)$$

⌊ ⌋ denotes the floor operator (as in practice an integer division is performed)



Types of convolutions:

- Valid convolution: using no padding
- Same convolution: output=input size

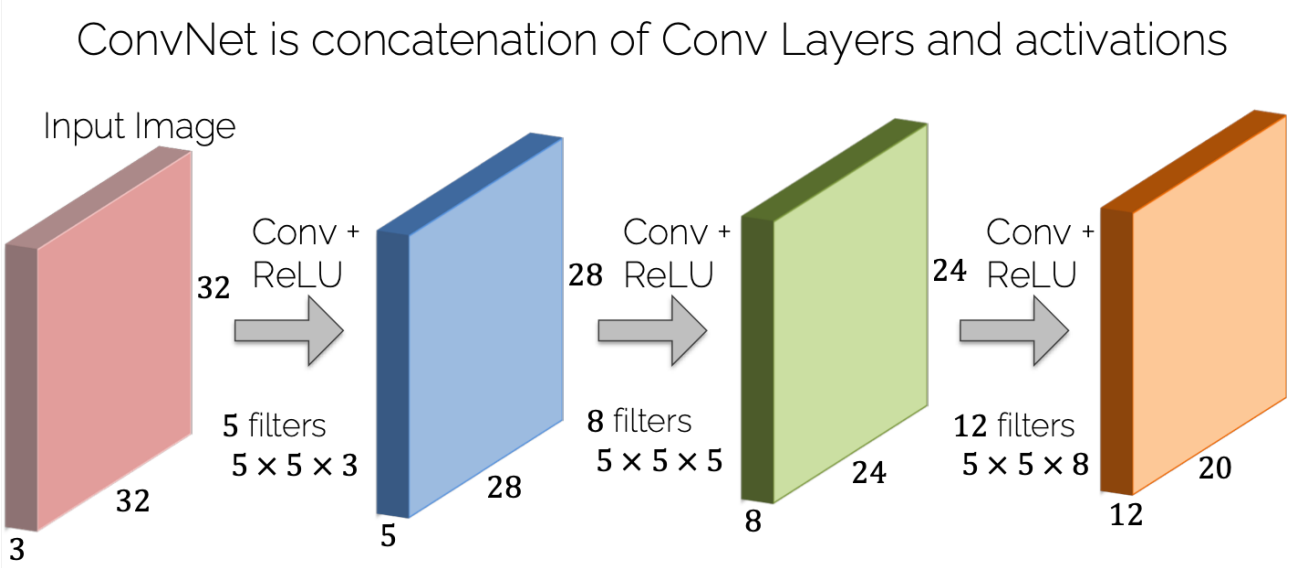
Set padding to $P = \frac{F-1}{2}$

Convolution Layers: Dimension

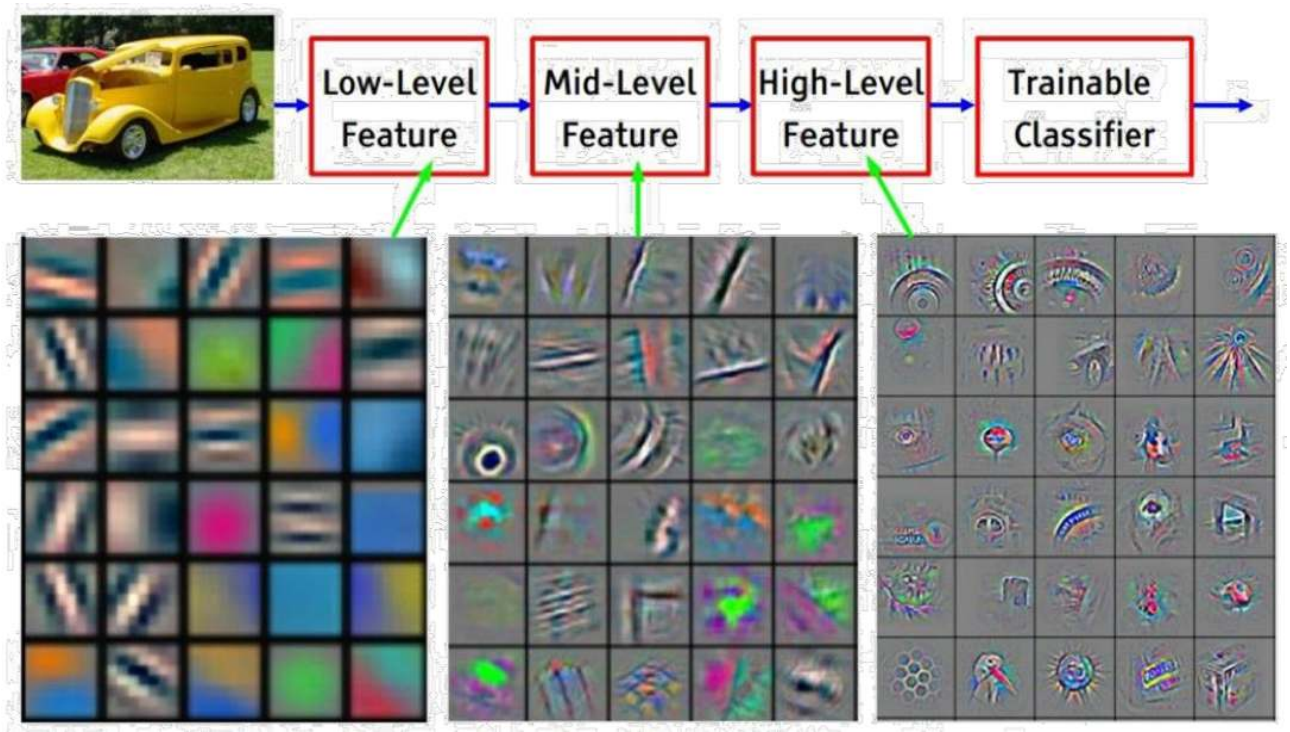
MIRAR EXEMPLE PÀG 63-65
MIRAR SEGÜENT EXEMPLE TAMBÉ PÀG 66-67

Convolutional Neural Network (CNN)

CNN Prototype

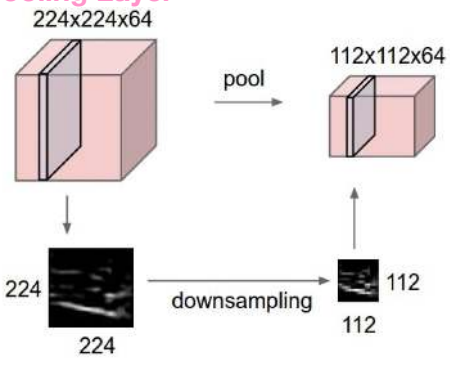


CNN Learned Filters



Pooling

Pooling Layer



Pooling Layer: Max Pooling

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

Max pool with 2×2 filters and stride 2

'Pooled' output

6	9
3	4

Pooling Layer

· Conv Layer = "Feature Extraction"
- Compute a fature in a given region

· Pooling Layer = " Feature Selection"
- Picks the strongest activation in a region

- Input is a volume of size $W_{in} \times H_{in} \times D_{in}$
- Two hyperparameters
  - Spatial filter extent F
  - Stride S

  } Filter count K and padding P make no sense here
- Output volume is of size $W_{out} \times H_{out} \times D_{out}$
  - $W_{out} = \frac{W_{in}-F}{S} + 1$
  - $H_{out} = \frac{H_{in}-F}{S} + 1$
  - $D_{out} = D_{in}$
- Does not contain parameters; e.g. it's fixed function

- Input is a volume of size $W_{in} \times H_{in} \times D_{in}$
- Two hyperparameters
  - Spatial filter extent F
  - Stride S
- Output volume is of size $W_{out} \times H_{out} \times D_{out}$
  - $W_{out} = \frac{W_{in}-F}{S} + 1$
  - $H_{out} = \frac{H_{in}-F}{S} + 1$
  - $D_{out} = D_{in}$
- Does not contain parameters; e.g. it's fixed function

Common settings:
$F = 2, S = 2$
$F = 3, S = 2$

Pooling Layer: Average Pooling

Single depth slice of input

3	1	3	5
6	0	7	9
3	2	1	4
0	2	4	3

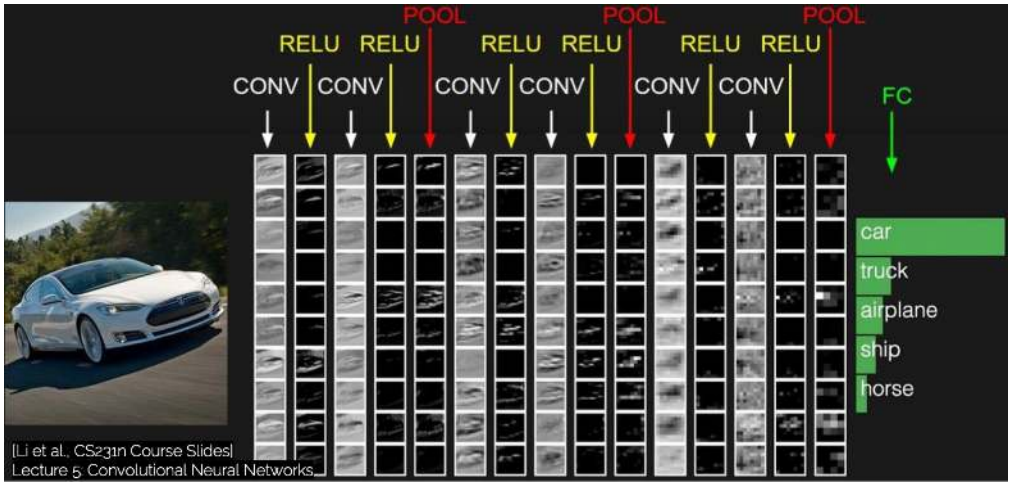
Average pool with 2×2 filters and stride 2

'Pooled' output

2.5	6
1.75	3

- Typically used deeper in the network

CNN Prototype



Finally Fully - Connected Layer

· Same as what we had in “ordinary” neural networks
  - Make the final decision with the extracted features from the convolutions
  - One or two FC layer typically

Convolutions vs Fully-Connected

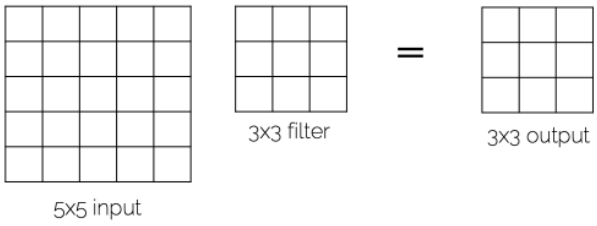
· In contrast of fully connected layers, we want to restrict the degrees of freedom.
  - FC is somewhat brute force
  - Convolutions are structured

· Sliding window to with the same filter parameters to extract image features
  - Concept of weight sharing
  - Extract same features independent of location

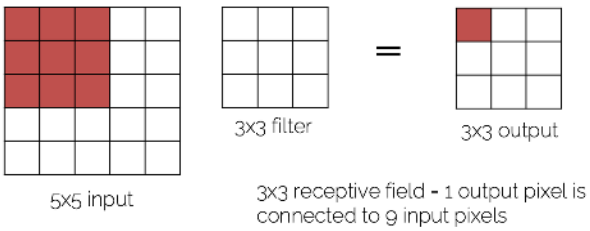
Receptive Field

Receptive Filed

- Spatial extent of the connectivity of a convolutional filter

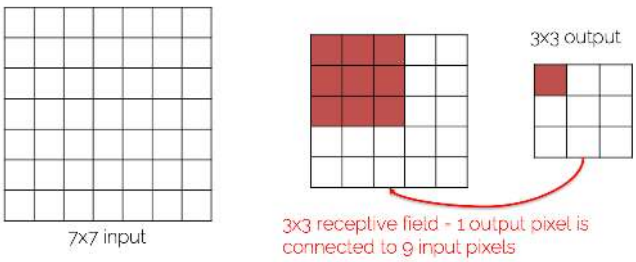


- Spatial extent of the connectivity of a convolutional filter

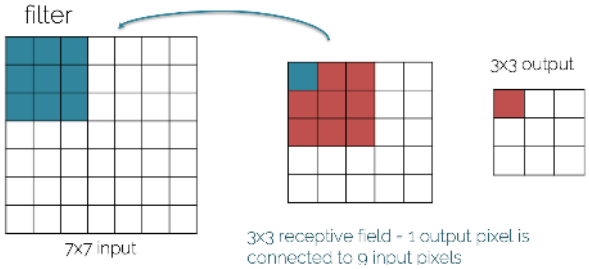


3x3 receptive field - 1 output pixel is connected to 9 input pixels

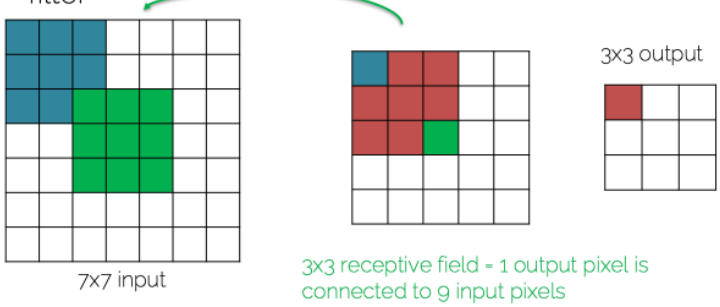
- Spatial extent of the connectivity of a convolutional filter



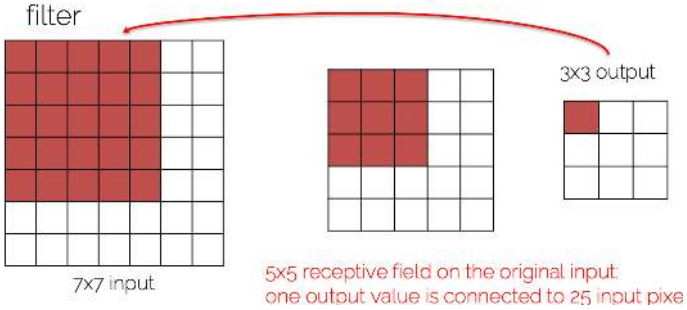
- Spatial extent of the connectivity of a convolutional filter



- Spatial extent of the connectivity of a convolutional filter



- Spatial extent of the connectivity of a convolutional filter

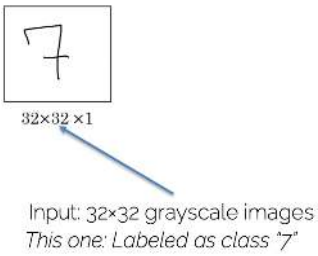


LECTURE 10: Convolutional Neural Networks II

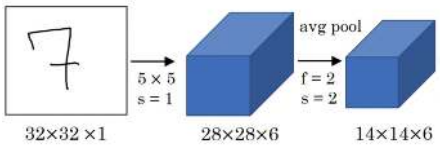
Classic Architectures

Le Net

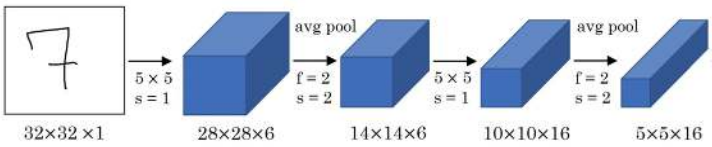
· Digit recognition: 10 classes



· At that time average pooling was used, now max pooling is much more common



· Again valid convolutions, how many filters?

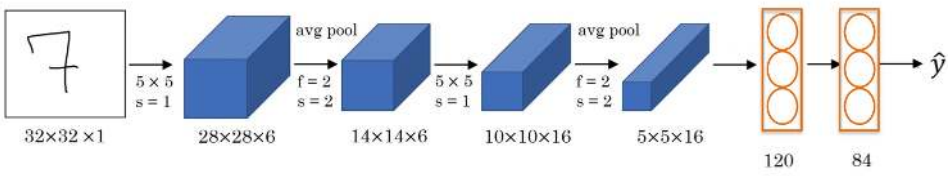


· Use of tanh/sigmoid activations —> not common now!

· Conv —> Pool —> Conv —> Pool —> Conv —> FC

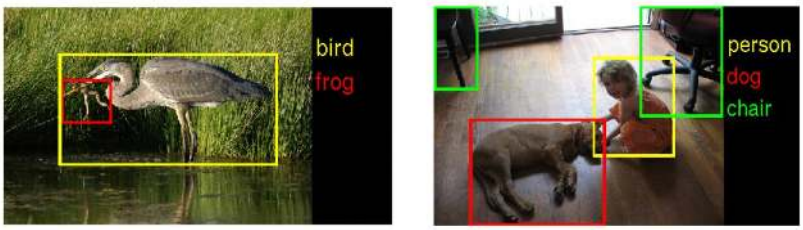
· As we go deeper: Width, Height go down Number of filters go up

60k parameters



Test Benchmarks

- ImageNet Dataset:
  ImageNet Large Scale Visual Recognition Competition (ILSVRC)



Common Performance Metrics

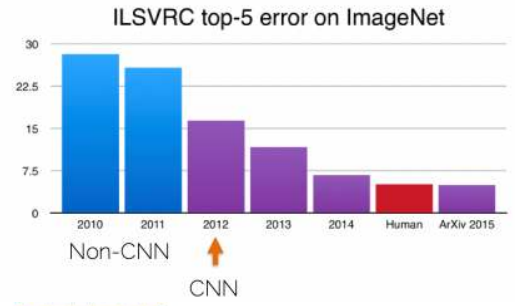
· Top-1 score: check if a sample's top class (i.e. the one with highest prob) is the same as its target label.

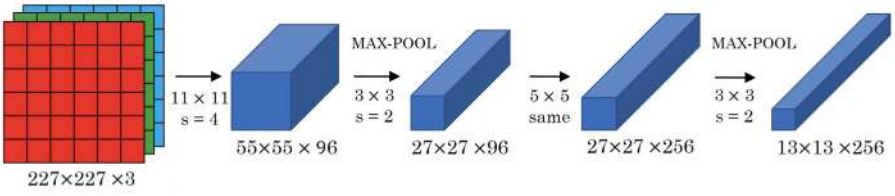
· Top-5 score: check if your label is in your 5 first predictions (i.e. predictions with 5 highest prob)

· —> Top-5 error: percentage of test samples for which the correct class was to in the top 5 predictions classes.

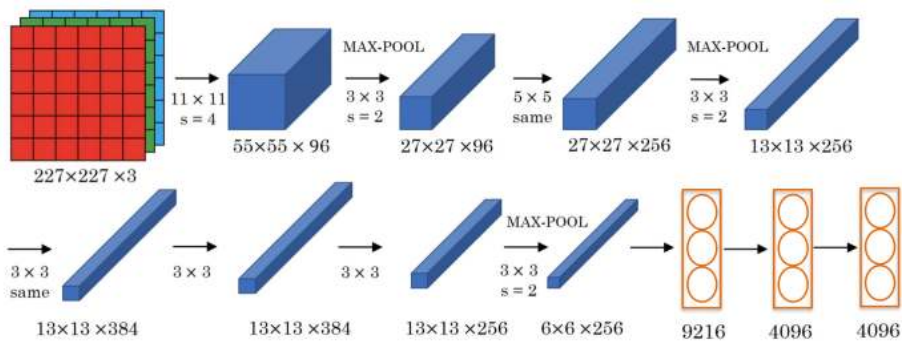
AlexNet

- Cut ImageNet error down in half





- Use of same convolutions
- As with LeNet: Width, Height ↓ Number of Filters ↑



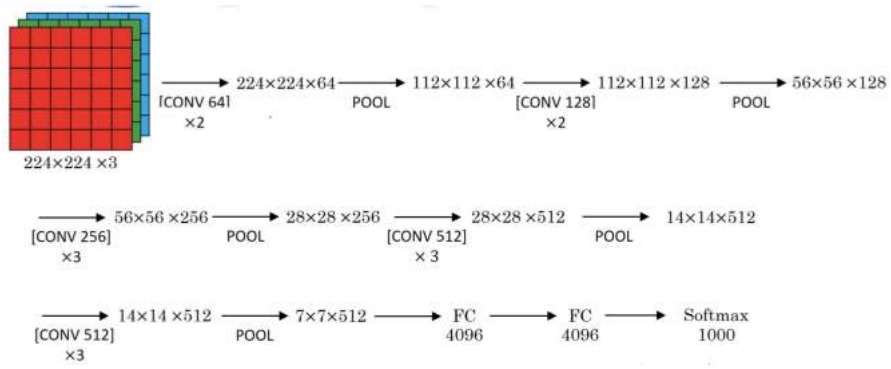
- Softmax for 1000 classes

· Similar to LeNet but much bigger (~1000 times)
· Use of ReLU instead of tanh/sigmoid

60M parameters

VGGNet

· Striving for simplicity
· CONV = 3x3 filters with stride 1, same convolutions
· MAXPOOL = 2x2 filters with stride 2



· Conv —> Pool —> Conv —> Pool —> Conv —> FC

· As we go deeper: Width, Height go down Number of Filters go up

· Called VGG-16: 16 layers that have weights

138M parameters

· Large but simplicity makes it appealing

· A lot of architectures were analyzed

Skip Connections

The Problem of Depth

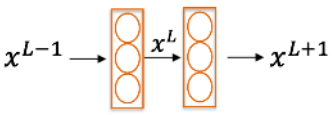
· As we add more and more layers, training becomes harder

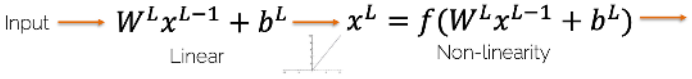
· Vanishing and exploding gradients

· how can we train very deep nets?

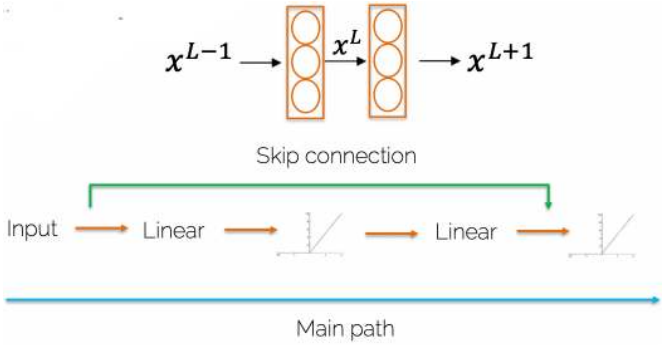
Residual Block

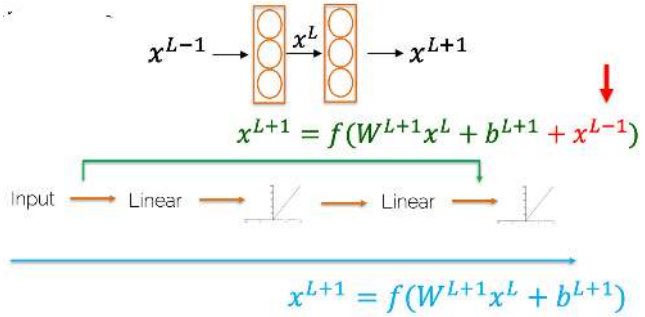
- Two layers

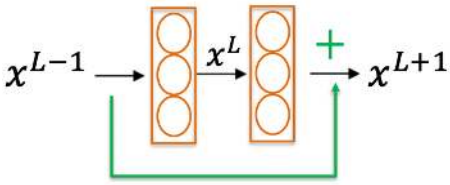




$\longrightarrow x^{L+1} = f(W^{L+1}x^L + b^{L+1})$



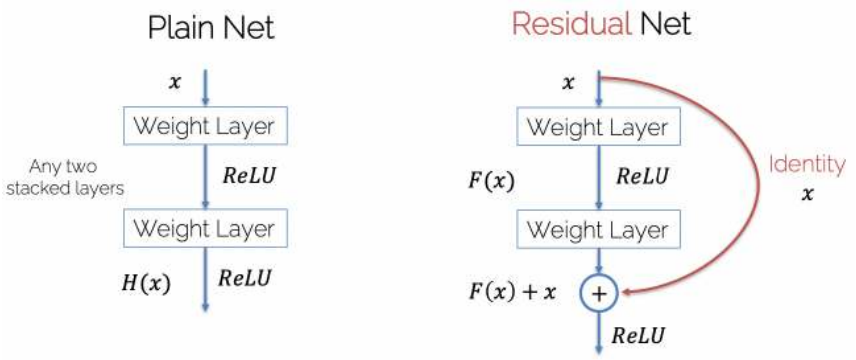




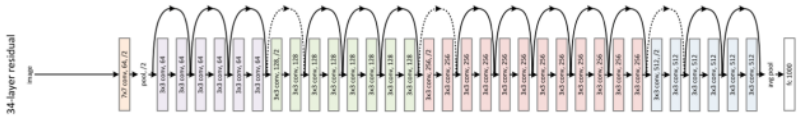
· Usually use same convolution since we need same dimensions

· Otherwise we need to convert the dimensions with a matrix of learned weights or zero padding

ResNet Block



ResNet

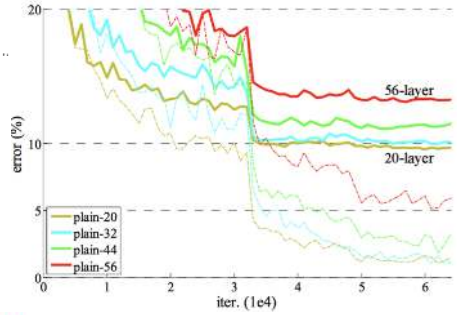


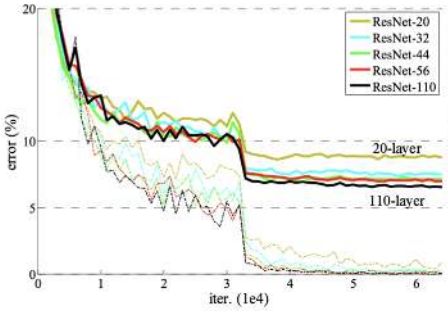
- Xavier/2 initialization
- SGD + Momentum (0.9)
- Learning rate 0.1, divided by 10 when plateau
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout

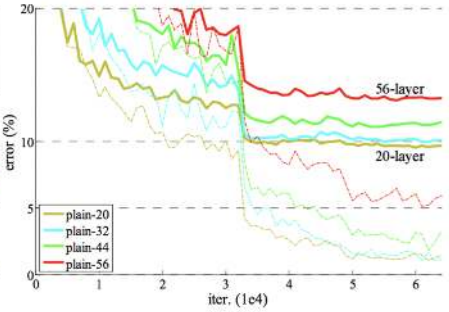
ResNet-152:
60M parameters

· If we make the network deeper, at some point performance starts to degrade

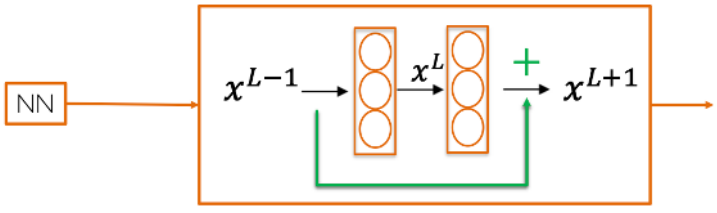
· If we make the network deeper, at some point Performance starts to degrade



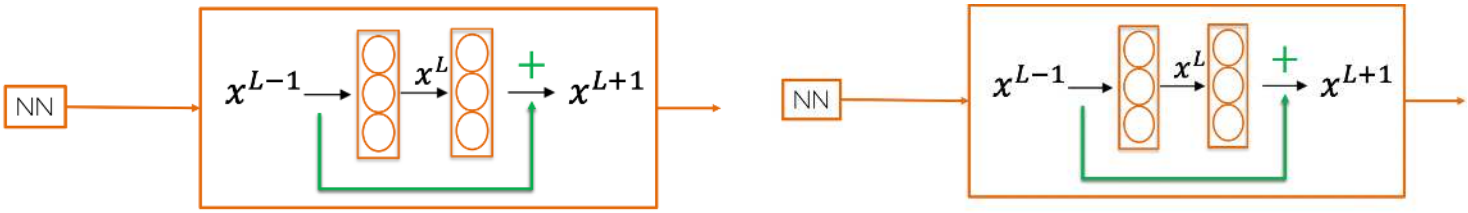




Why does ResNet work?



· How is this block really affecting me?



$$x^{L+1} = f(W^{L+1}x^{L} + b^{L+1} + x^{L-1})$$

~zero ~zero

$$x^{L+1} = f(x^{L-1})$$

· We kept the same values and added a non-linearity $x^{L+1} = f(x^{L-1})$

· The identity is easy for the residual block to learn

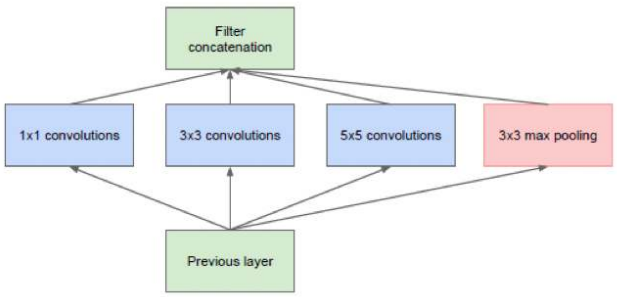
· Guaranteed it will not hurt performance, can only improve

1x1 Convolutions

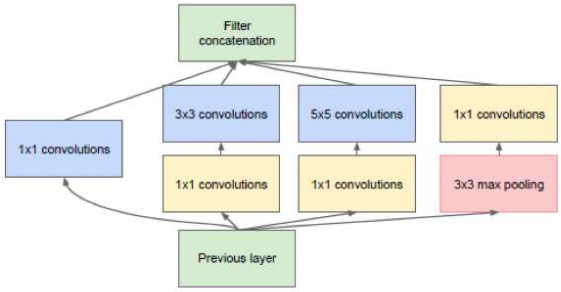
MIRAR L'EXEMPLE PÀG 57-64

Inception Layer

Inception Layer

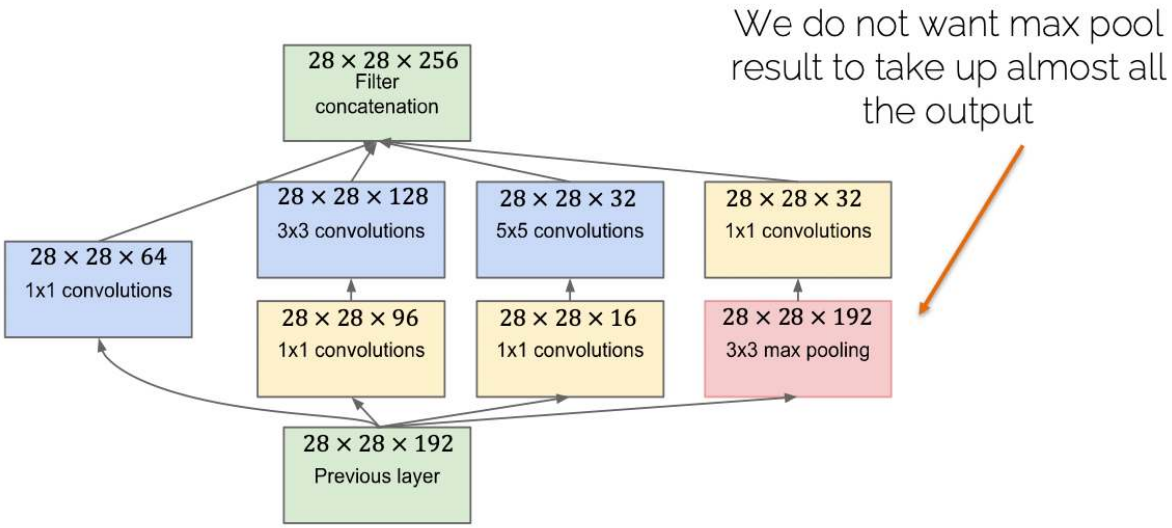


(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Inception Layer: Dimensions



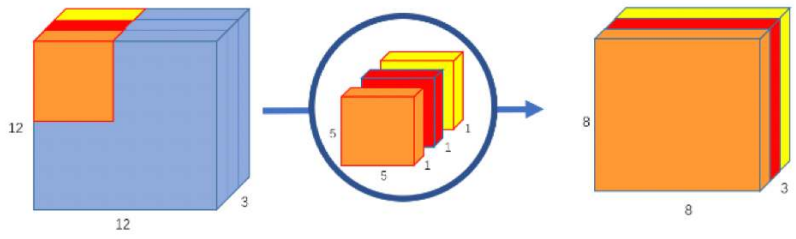
Xception Net

· “ extreme version of Inception”: applying (modified) Depthwise Separable Convolutions instead of normal convolutions.

· 36 conv layers, structured into several modules with skip connections.

· outperforms Inception Net V3

Depth-wise separable convolutions

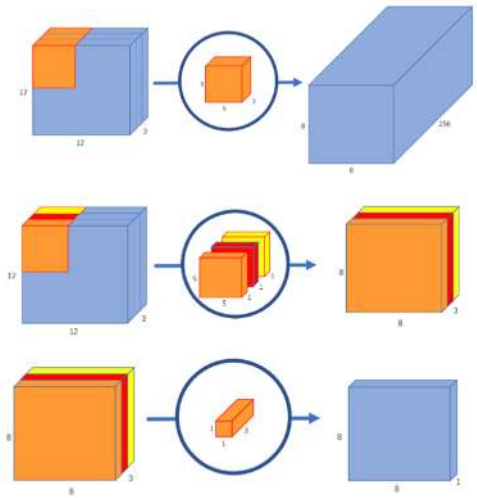


`class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)`

`class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, groups=3)`

Filterers are applied only at a certain depth of the features. Normal convolutions have groups set to 1, the convolutions used in this image have groups set to 3.

But the depth size is always the same!



Original convolution
256 kernels of size 5x5x3

Multiplications:
256x5x5x3 x (8x8 locations) = 1.228.800

Depth-wise convolution
3 kernels of size 5x5x1

Multiplications:
5x5x3 x (8x8 locations) = 4800

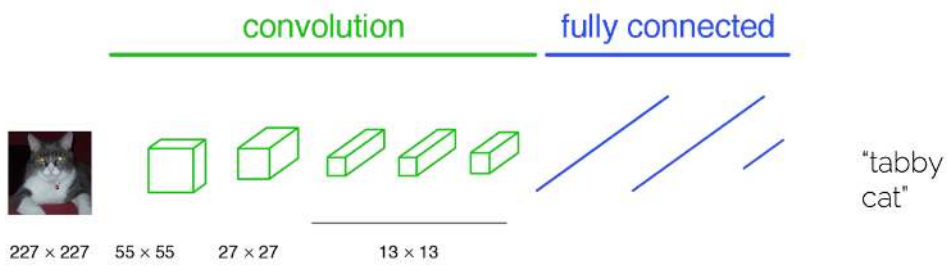
Less computation!

1x1 convolution
256 kernels of size 1x1x3

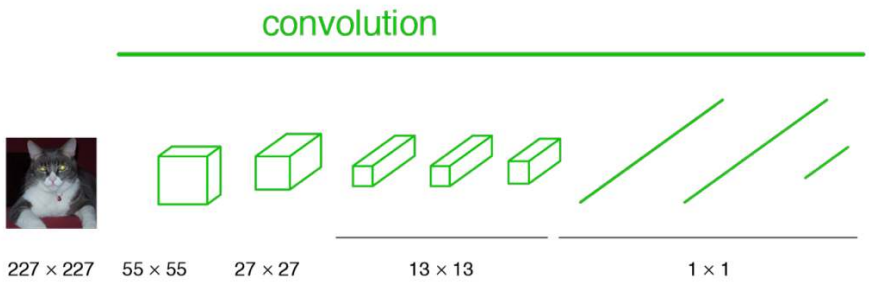
Multiplications:
256x1x1x3x (8x8 locations) = 49152

Fully Convolutional Network

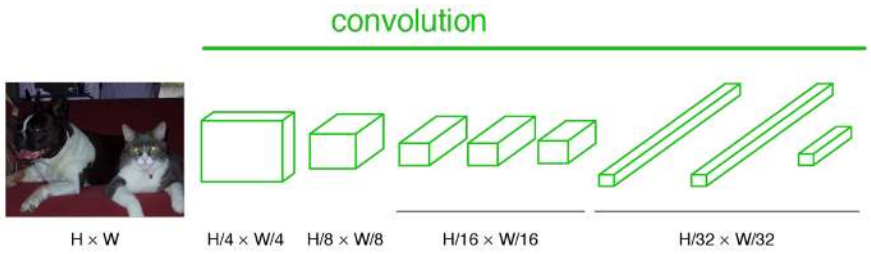
Classification Network



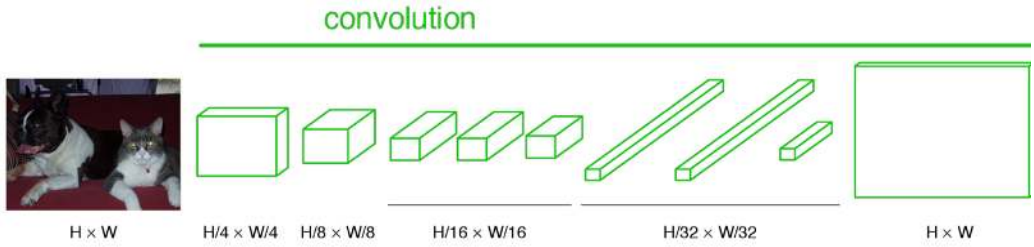
FCN: Becoming Fully Convolutional



Convert fully connected layers to convolutional layers!

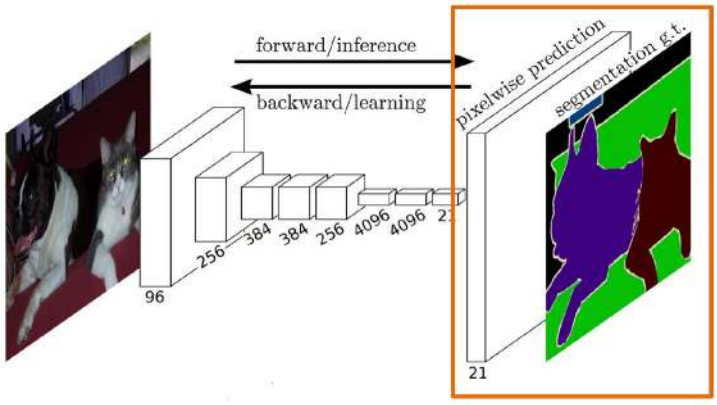


FCN: Upsampling Output



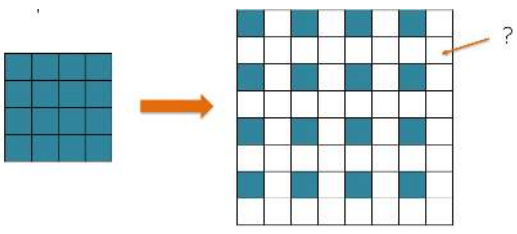
Semantic Segmentation

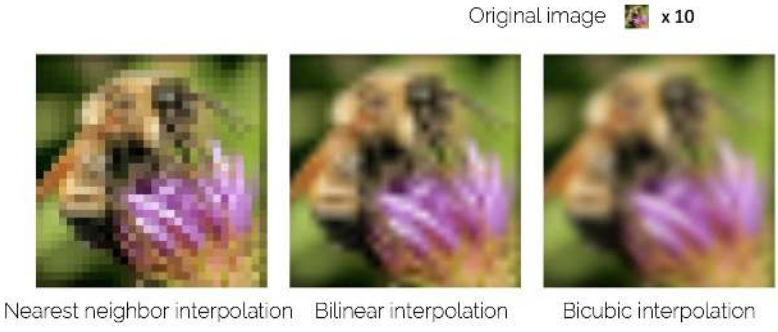
How do we go back to the input size?



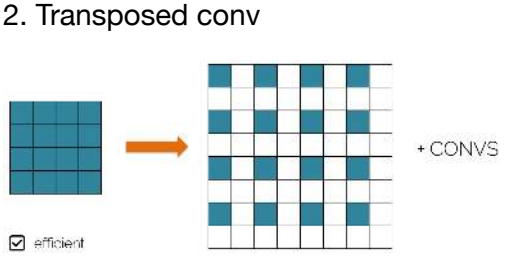
Types of Upsampling

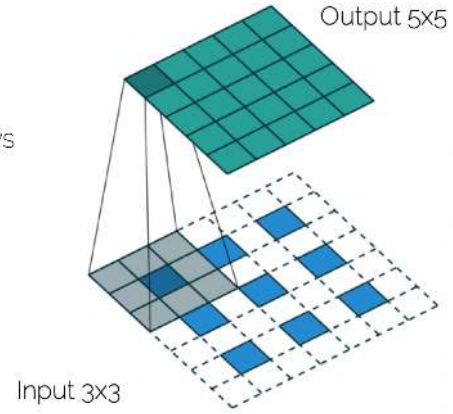
1. Interpolation





2. Transposed conv





- Unpooling
- Convolutional filter (learned)
- Also called up-convolution

(Never deconvolution)

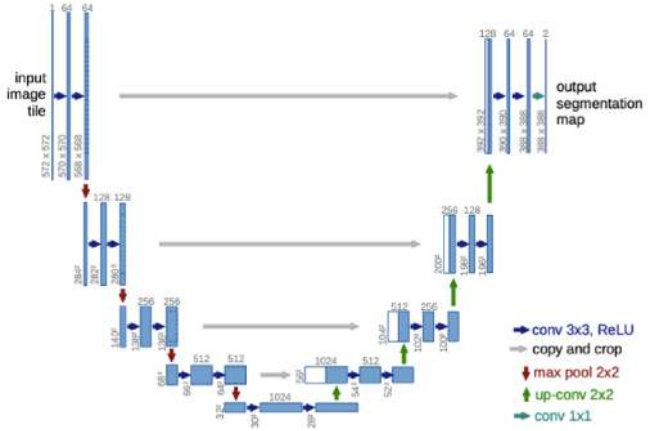
Refined Outputs

· If one does a cascade of unpooling + conv operations, we get to the encoder-decoder architecture.

· Even more refined: Autoencoders with skip connections (aka U-Net)

U-Net

U-Net architecture: Each blue box is a multichannel feature map. Number of channels denoted at the top of the box. Dimensions at the top of the box. White boxes are the copied feature maps.



U-Net: Encoder

Left side: Contraction Path (Encoder)

· Captures context of the images

· Follows typical architecture of a CNN:
- Repeated applications of 2 unpadded 3x3 convolutions.
- Each followed by ReLU activation
- 2x2 maxpooling operation with stride 2 for downsampling
- At each downsampling step, # channels is doubled

—> As before: Height, Width go down, Depth go up

U-Net:Decoder

Right side: Expansion path (Decoder):

· Upsampling to recover spatial locations for assigning class labels to each pixel
- 2x2 up-convolution that halves number of input channels
- Skip Connections: outputs oof up-convolutions are concatenated with feature maps from encoder.

- Followed by 2 ordinary 3x3 convs
- Final layer : 1x1 conv to map 64 channels to # classes

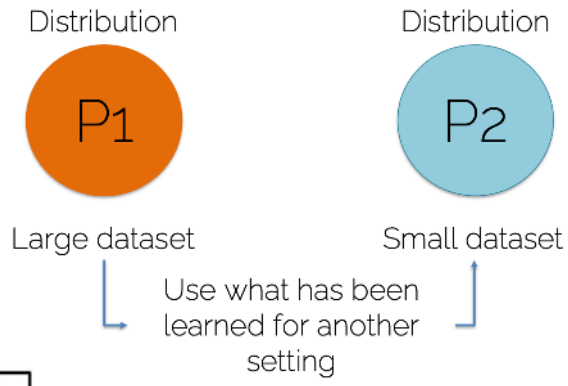
· Height, Width goes up, Depth goes down.

LECTURE 11: Recurrent Neural Networks and Transformers

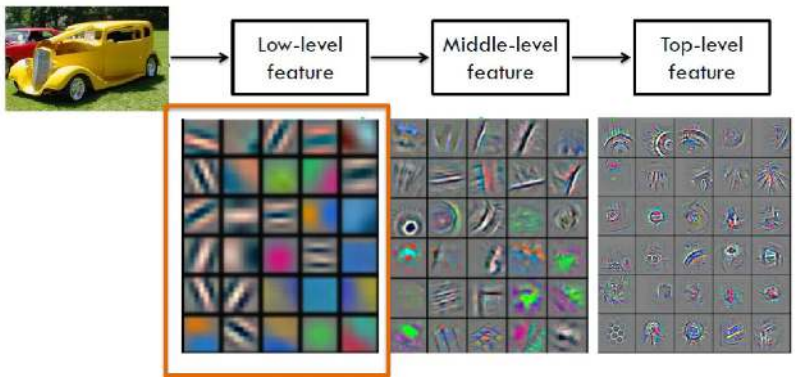
Transfer Learning

Transfer Learning

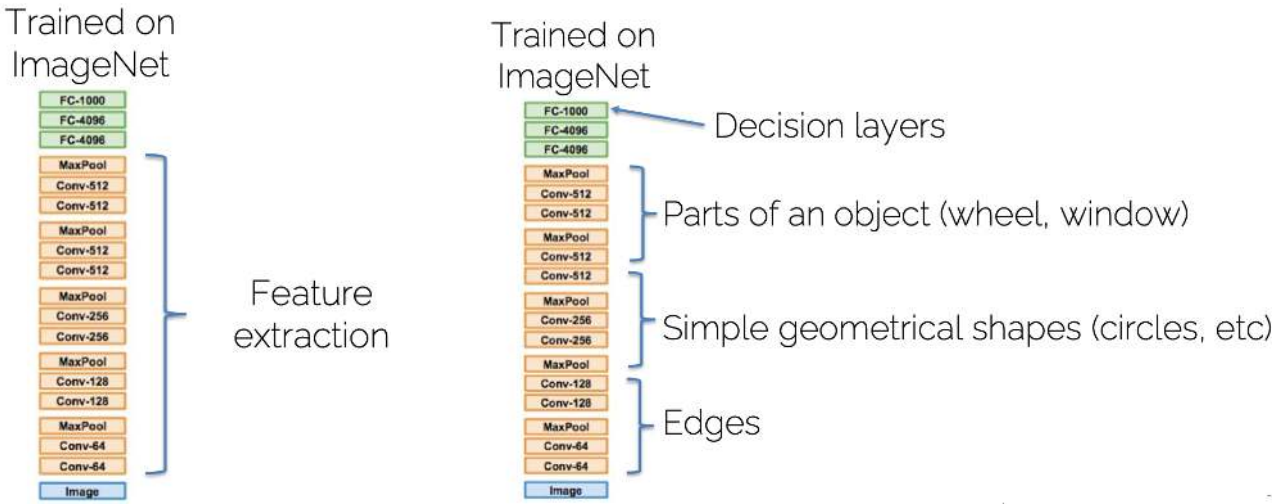
· Training your own model can be difficult with limited data and other resources.
e.g.,
· It is a laborious task to manually annotate your own training dataset
· Why not reuse already pre-trained models?

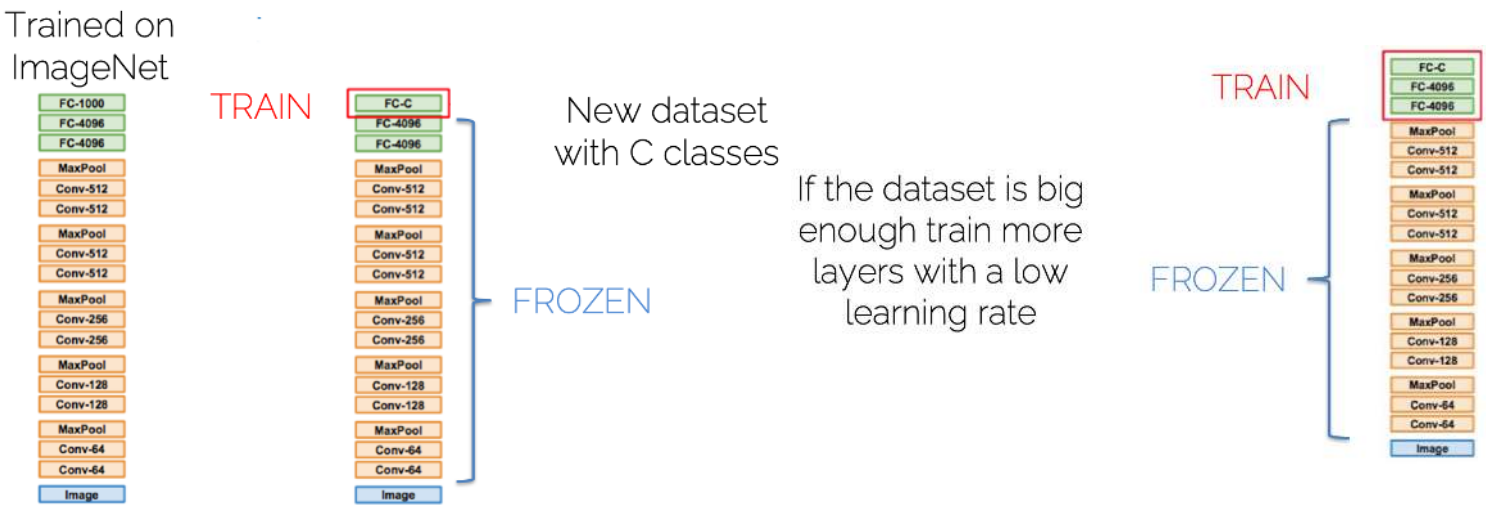


Transfer Learning for Images



Transfer Learning





When Transfer Learning Makes Sense

· When task T1 and T2 have the same input (e.g. an RGB image).

· When you have more data for task T1 than for task T2.

· When the low-level features for T1 could be useful to learn T2.

Representation Learning

Learning Good Features

· Good features are essential for successful machine learning

· (Supervised) deep learning depends on training data used: input/target labels

· Change in inputs (noise, irregularities, etc) can result in drastically different results

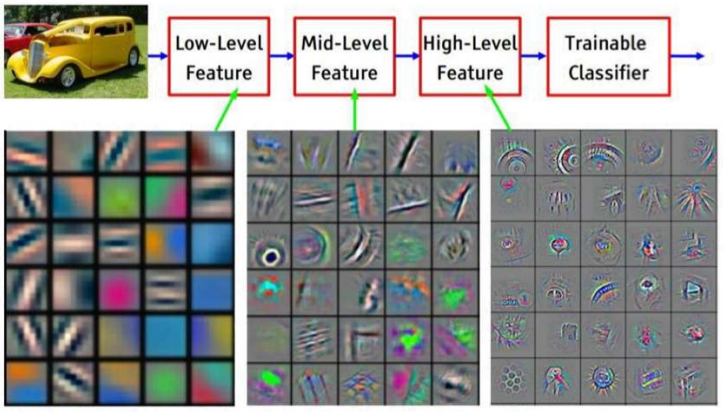
Representation Learning

· Allows for discovery of representation required for various tasks

· Deep representation learning: model maps input X to output Y

Deep Representation Learning

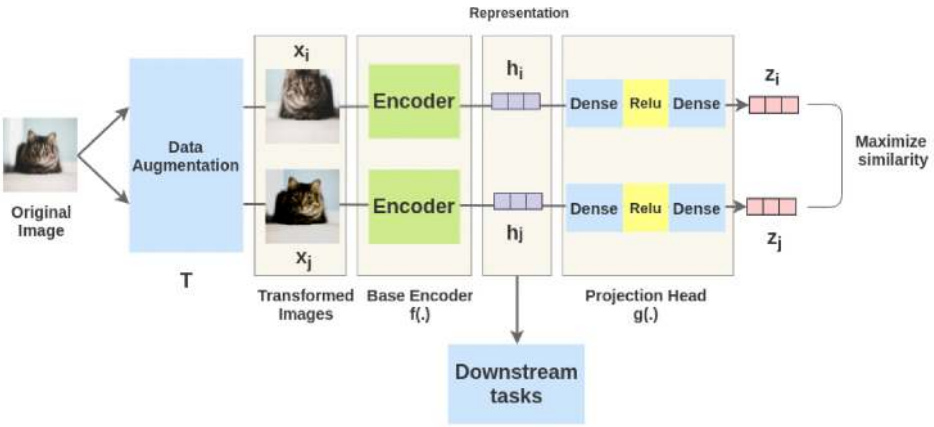
· Intuitively, deep networks learn multiple levels of abstraction



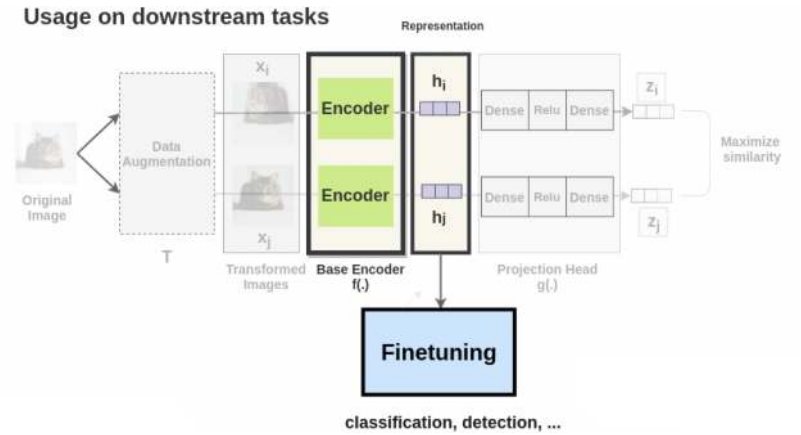
How to Learn Good Features?

· Determine desired features invariances

· Teach machines to distinguish between similar and dissimilar things.



How to Learn Good Features?



Transfer and Representation Learning

· Transfer learning can be done via representation learning

· Effectiveness of representation learning often demonstrated by transfer learning performance (but also other factors, e.g., smoothness of the manifold)

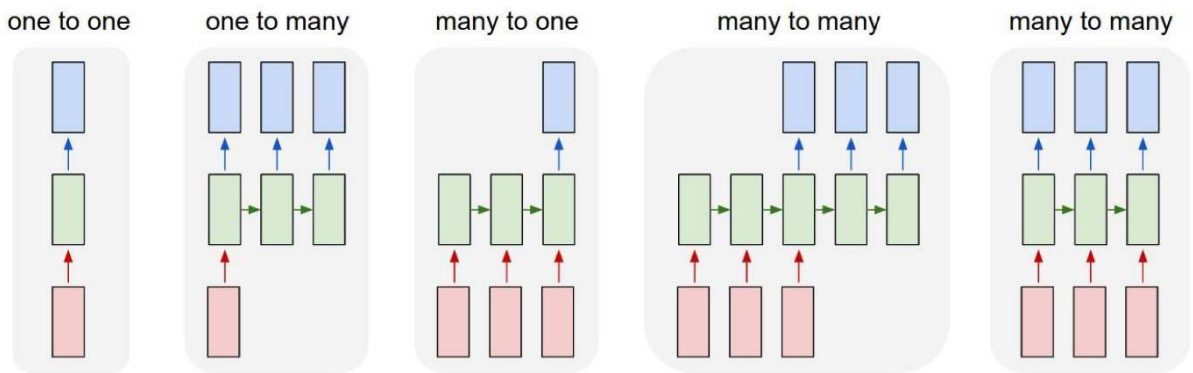
Recurrent Neural Networks

Processing Sequences

· Recurrent neural networks process sequence data

· Input/output can be sequences

RNNs are Flexible



ONE TO ONE —> Classical neural networks for image classification

ONE TO MANY —> image captioning

MANY TO ONE —> Language recognition

MANY TO MANY —> Machine translation

MANY TO MANY —> Event classification

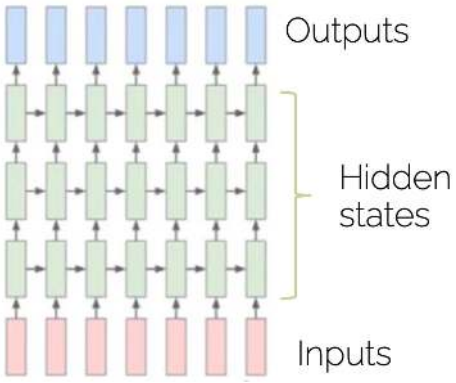
Basic Structure of an RNN

- Multi-layer RNN

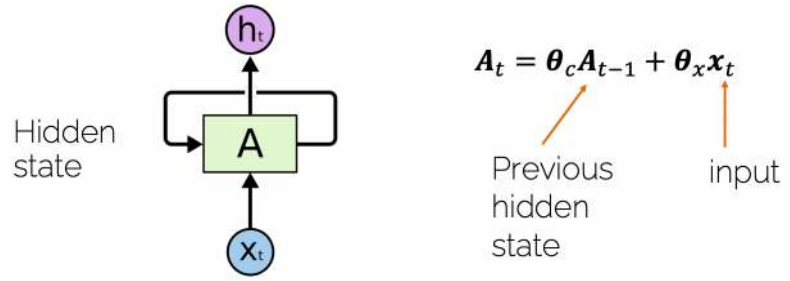
The hidden state will have its own internal dynamics

↓

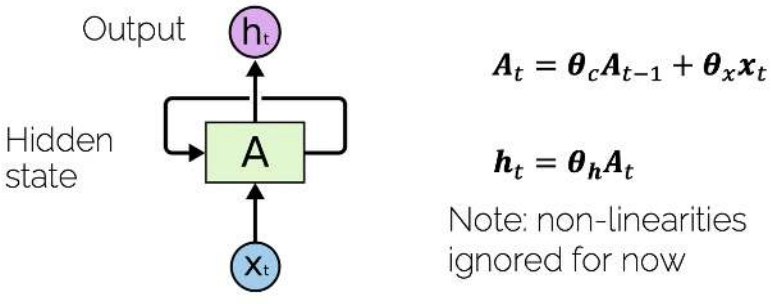
More expressive model!



· We want to have notion of “time” or “sequence”



θc, θx —> Parameters to be learned

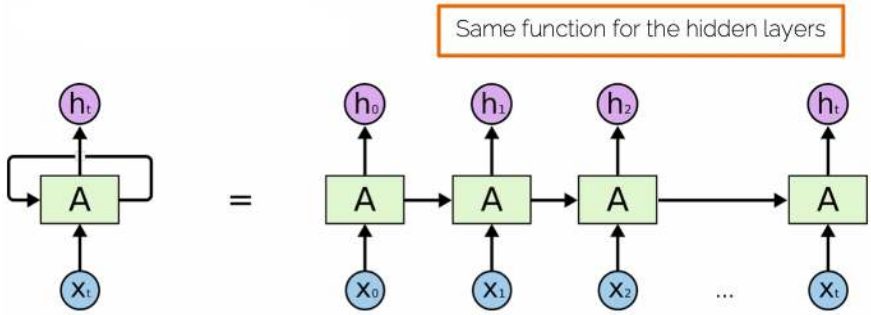


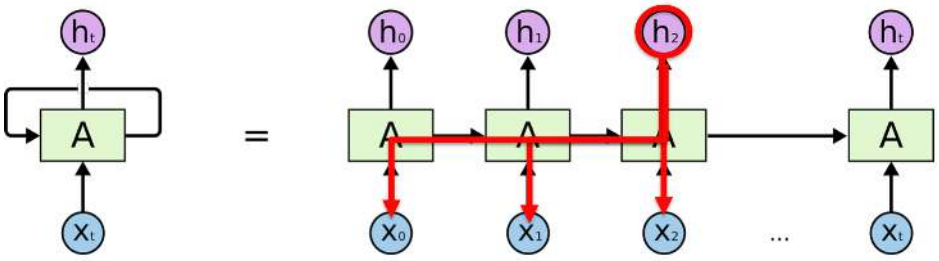
$A_t = \theta_c A_{t-1} + \theta_x x_t$

$h_t = \theta_h A_t$

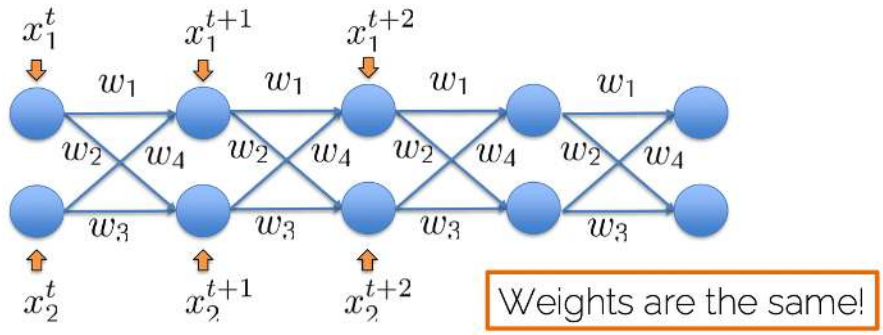
Same parameters for each time step = generalization!

· Unrolling RNNs



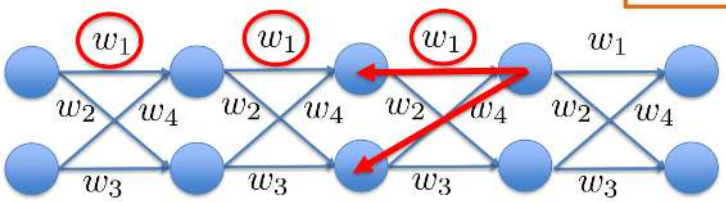


· Unrolling RNNs as feedforward nets



Weights are the same!

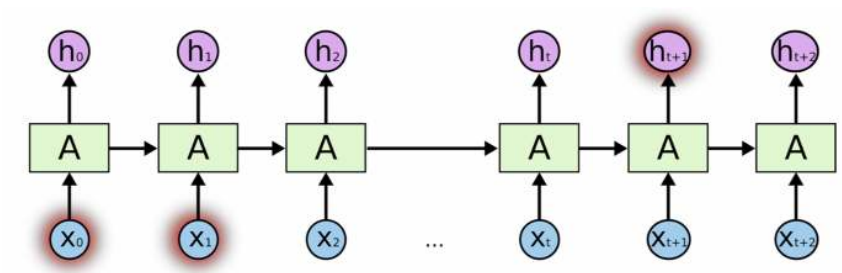
Chain rule



All the way to $t = 0$

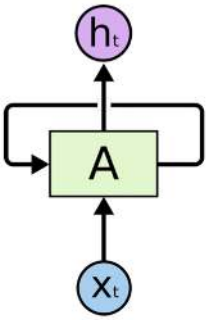
Add the derivatives at different times for each weight

Long-term Dependencies



I moved to Germany ... so I speak German fluently.

- Simple recurrence $\boldsymbol{A_t = \theta_c A_{t-1} + \theta_x x_t}$
- Let us forget the input $\boldsymbol{A_t = \theta_c{}^t A_0}$



Same weights are multiplied over and over

· Simple recurrence $\boldsymbol{A_t = \theta_c{}^t A_0}$

What happens to small weights —> Vanishing gradient

What happens to large weights? —> Exploding gradients

· If θ admits eigendecomposition

$$\boldsymbol{\theta = Q\Lambda Q^T}$$

Matrix of eigenvectors

Diagonal of this matrix are the eigenvalues

· Orthogonal θ allows us to simplify the recurrence $\boldsymbol{A_t = Q\Lambda^t Q^T A_0}$

· Simple recurrence $\boldsymbol{A_t = Q\Lambda^t Q^T A_0}$

What happens to eigenvalues with magnitude less than one? —> Vanishing gradient

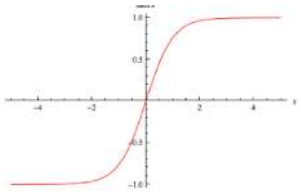
What happens to eigenvalues with magnitude larger than one? —> Exploding gradient (Gradient clipping)

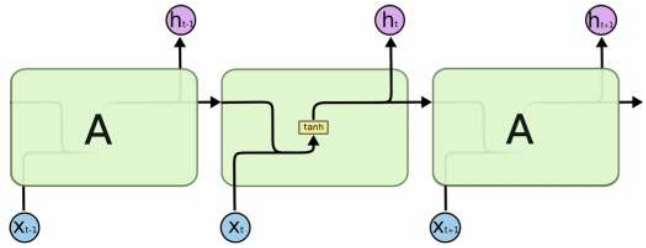
· Simple recurrence $\boldsymbol{A_t = \theta_c{}^t A_0}$ Let us just make a matrix with eigenvalues = 1 (Allow the cell to maintain its "state"

Vanishing Gradient

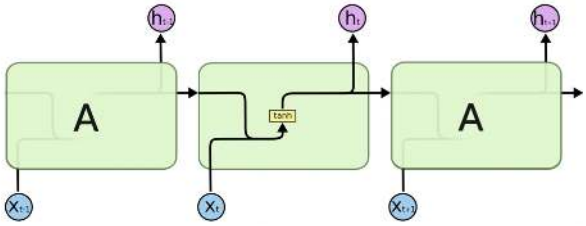
· 1. From the weights $\boldsymbol{A_t = \theta_c{}^t A_0}$

· 2. From the activation functions ($tanh$)





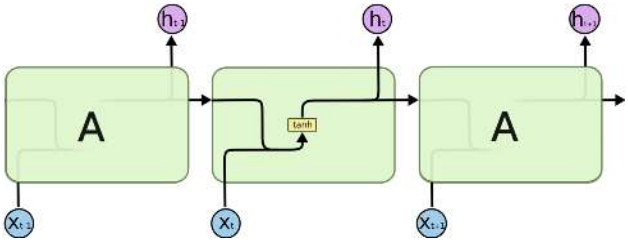
- 1. From the weights $\boldsymbol{A_t = \theta^t A_0}$ (1)
- 2. From the activation functions ($tanh$) ?

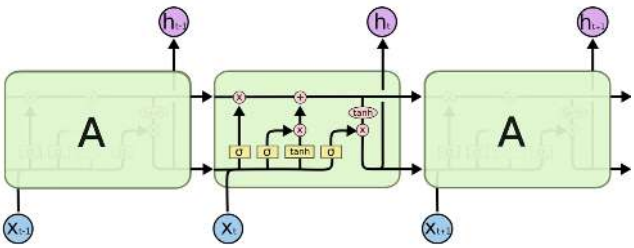


Long Short Term Memory

Long-Short Term Memory Units

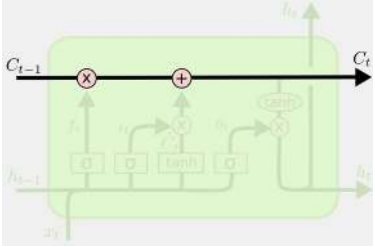
· Simple RNN has tanh as non-linearity

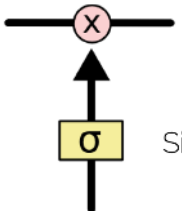




· Key ingredients

· Cell = transport the information through the unit



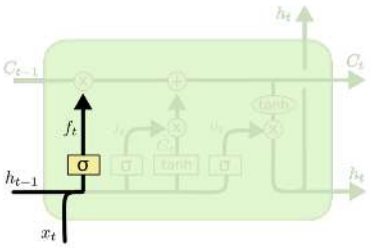


Sigmoid

· Gate = remove or add information to the cell state

LSTM: Step by Step

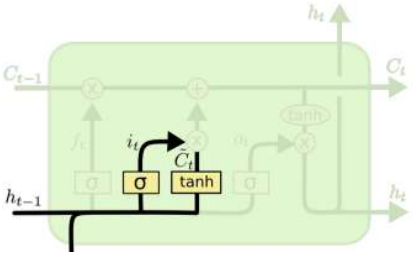
- Forget gate $\boldsymbol{f_t = sigm(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)}$



Decides when to erase the cell state

Sigmoid - output between **0** (forget) and **1** (keep)

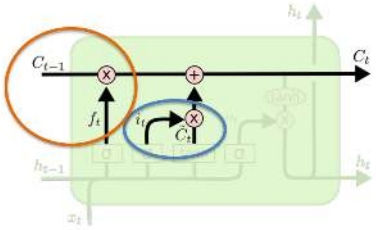
- Input gate $\boldsymbol{i_t = sigm(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)}$



Decides which values will be updated

New cell state, output from a $\boldsymbol{\tanh(-1,1)}$

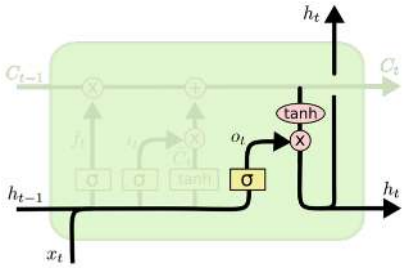
- Element-wise operations



$$\boldsymbol{C_t = f_t \odot C_{t-1} + i_t \odot g_t}$$

Previous states — Current state

- Output gate $\boldsymbol{h_t = o_t \odot \tanh(C_t)}$



Decides which values will be outputted

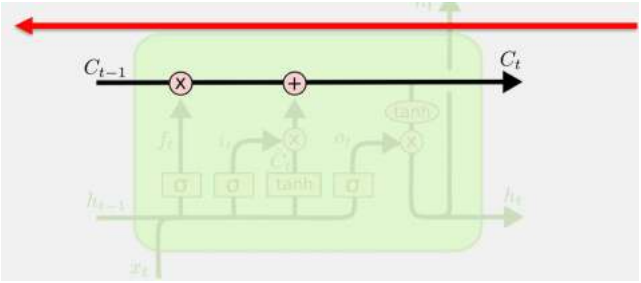
Output from a $\boldsymbol{\tanh(-1,1)}$

- Forget gate $\boldsymbol{f_t = sigm(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)}$
- Input gate $\boldsymbol{i_t = sigm(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)}$
- Output gate $\boldsymbol{o_t = sigm(\theta_{xo} x_t + \theta_{ho} h_{t-1} + b_o)}$
- Cell update $\boldsymbol{g_t = tanh(\theta_{xg} x_t + \theta_{hg} h_{t-1} + b_g)}$
- Cell $\boldsymbol{C_t = f_t \odot C_{t-1} + i_t \odot g_t}$
- Output $\boldsymbol{h_t = o_t \odot \tanh(C_t)}$

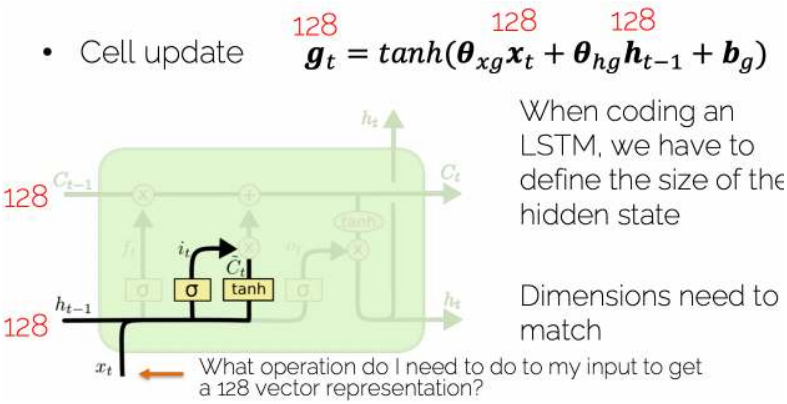
Learned through backpropagation

LSTM

· Highway fro the gradient to flow



LSTM: Dimensions



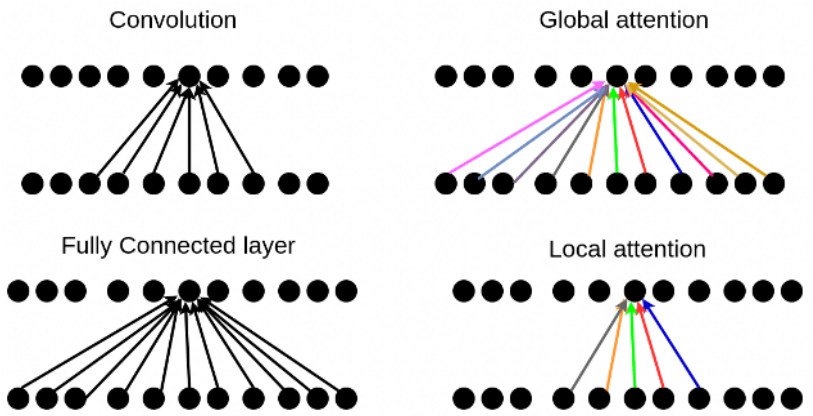
LSTM in code

```python
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
  """
  Forward pass for a single timestep of an LSTM.

  The input data has dimension D, the hidden state has dimension H, and we use
  a minibatch size of N.

  Inputs:
  - x: Input data, of shape (N, D)
  - prev_h: Previous hidden state, of shape (N, H)
  - prev_c: previous cell state, of shape (N, H)
  - Wx: Input-to-hidden weights, of shape (D, 4H)
  - Wh: Hidden-to-hidden weights, of shape (H, 4H)
  - b: Biases, of shape (4H,)

  Returns a tuple of:
  - next_h: Next hidden state, of shape (N, H)
  - next_c: Next cell state, of shape (N, H)
  - cache: Tuple of values needed for backward pass.
  """
  next_h, next_c, cache = None, None, None

  N, H = prev_h.shape
  # 1
  a = np.dot(x, Wx) + np.dot(prev_h, Wh) + b

  # 2
  ai = a[:, :H]
  af = a[:, H:2*H]
  ao = a[:, 2*H:3*H]
  ag = a[:, 3*H:]

  # 3
  i = sigmoid(ai)
  f = sigmoid(af)
  o = sigmoid(ao)
  g = np.tanh(ag)

  # 4
  next_c = f * prev_c + i * g

  # 5
  next_h = o * np.tanh(next_c)

  cache = i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h

  return next_h, next_c, cache
```

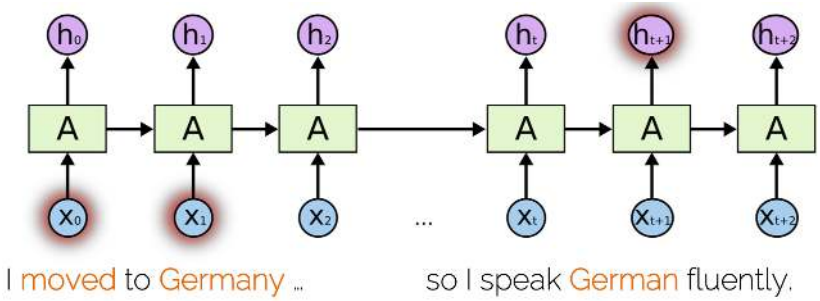
```python
def lstm_step_backward(dnext_h, dnext_c, cache):
  """
  Backward pass for a single timestep of an LSTM.

  Inputs:
  - dnext_h: Gradients of next hidden state, of shape (N, H)
  - dnext_c: Gradients of next cell state, of shape (N, H)
  - cache: Values from the forward pass

  Returns a tuple of:
  - dx: Gradient of input data, of shape (N, D)
  - dprev_h: Gradient of previous hidden state, of shape (N, H)
  - dprev_c: Gradient of previous cell state, of shape (N, H)
  - dWx: Gradient of input-to-hidden weights, of shape (D, 4H)
  - dWh: Gradient of hidden-to-hidden weights, of shape (H, 4H)
  - db: Gradient of biases, of shape (4H,)
  """
  dx, dh, dc, dWx, dWh, db = None, None, None, None, None, None

  i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h = cache

  # backprop into step 5
  do = np.tanh(next_c) * dnext_h
  dnext_c += o * (1 - np.tanh(next_c) ** 2) * dnext_h

  # backprop into 4
  df = prev_c * dnext_c
  dprev_c = f * dnext_c
  di = g * dnext_c
  dg = i * dnext_c

  # backprop into 3
  dai = sigmoid(ai) * (1 - sigmoid(ai)) * di
  daf = sigmoid(af) * (1 - sigmoid(af)) * df
  dao = sigmoid(ao) * (1 - sigmoid(ao)) * do
  dag = (1 - np.tanh(ag) ** 2) * dg

  # backprop into 2
  da = np.hstack((dai, daf, dao, dag))

  # backprop into 1
  db = np.sum(da, axis = 0)
  dprev_h = np.dot(Wh, da.T).T
  dWh = np.dot(prev_h.T, da)
  dx = np.dot(da, Wx.T)
  dWx = np.dot(x.T, da)

  return dx, dprev_h, dprev_c, dWx, dWh, db
```

Attention

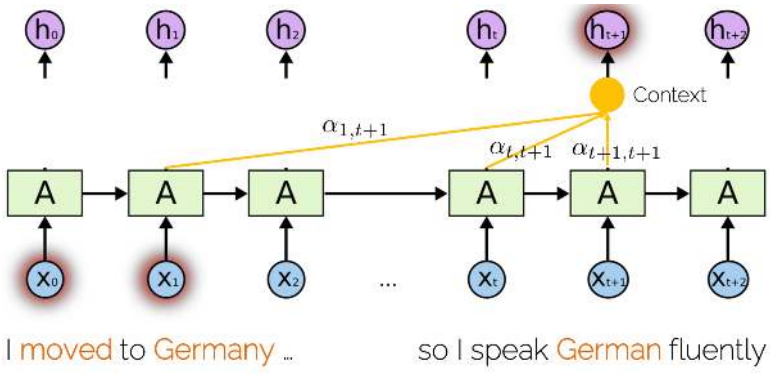
Attention vs convolution



Long-Term Dependencies



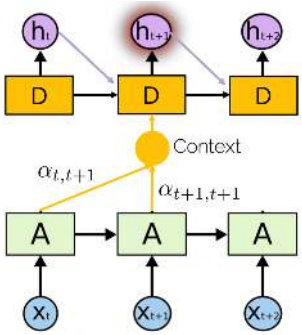
Attention: Intuition



Attention: Architecture

· A decoder processes the information

· Decoder takes as inout:
- Previous decoder hidden state
- Previous output
- Attention

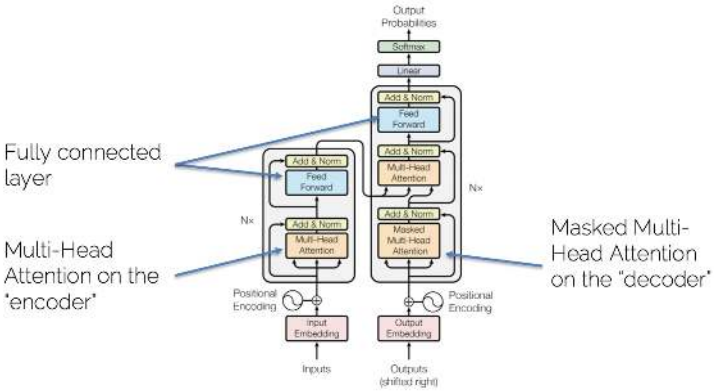


Transformers

Deep Learning Revolution

	Deep Learning	Deep Learning 2.0
Main idea	Convolution	Attention
Field invented	Computer vision	NLP
Started	NeurIPS 2012	NeurIPS 2017
Paper	AlexNet	Transformers
Conquered vision	Around 2014-2015	Around 2020-2021
~~Replaced~~ (Augmented)	Traditional ML/CV	CNNs, RNNs

Transformers

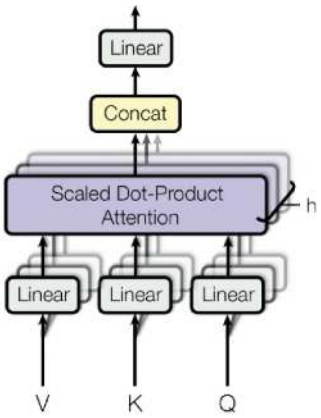


Multi-Head Attention

Intuition: Take the query Q, find the most similar key K , and then find the value V that corresponds to the key.

In other words, learn V, K, Q where:
V: here is a bunch of interesting things.
K: here is how we can index some things.
Q: I would like to know this interesting thing.

Loosely connected to Neural Turing Machines



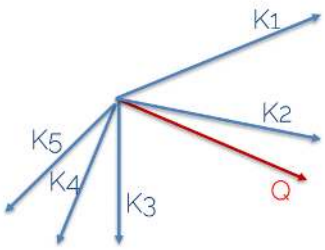
Index the values via a differentiable operator.

Multiply queries with keys

Get the values

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

To train them well, divide by $\sqrt{d_k}$, "probably" because for large values of the key's dimension, the dot product grows large in magnitude, pushing the softmax function into regions where it has extremely small gradients.



Values
V1
V2
V3
V4
V5

QK^T Essentially, dot product between (<Q,K1>), (<Q,K2>), (<Q,K3>), (<Q,K4>), (<Q,K5>).

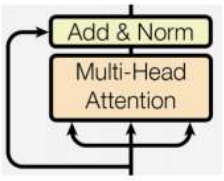
$\textbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$ Is simply inducing a distribution over the values. The larger a value is, the higher is its softmax value. Can be interpreted as a differentiable soft indexing.

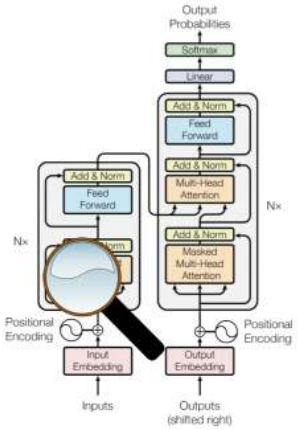
Values
V1
V2
V3
V4
V5

Selecting the value V where the network needs to attend..

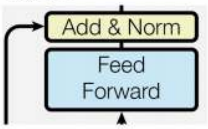
Transformers - a closer look

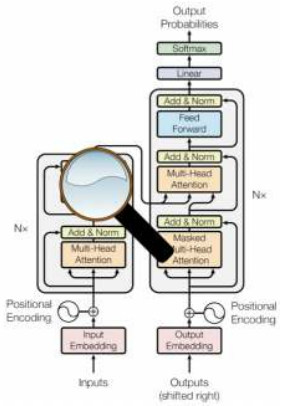
K parallel attention heads.



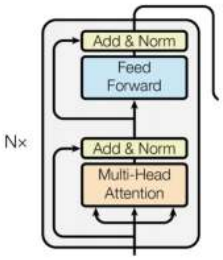


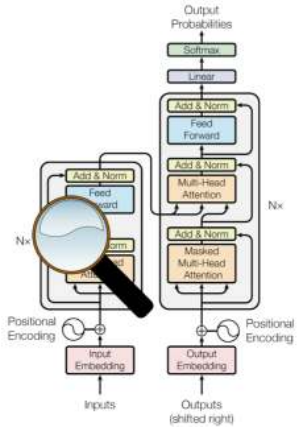
Good old fully-connected layers.



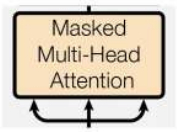


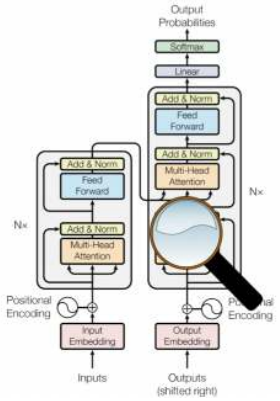
N layers of attention followed by FC



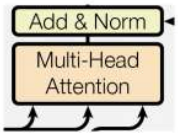


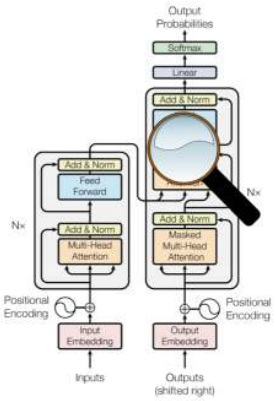
Same as multi-head attention, but masked. Ensures that the predictions for position i can depend only on the known outputs at positions less than i.



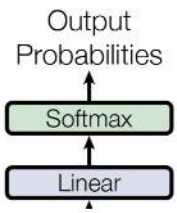


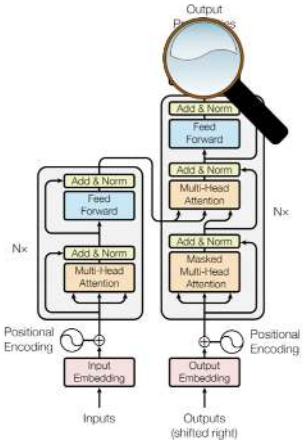
Multi-headed attention between encoder and the decoder.





Projection and prediction.

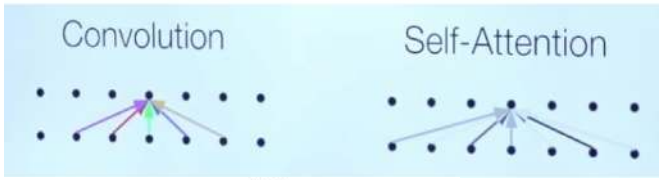




What is missing from self-attention?

· Convolution: a different linear transformation for each relative position. Allows you to distinguish what information came from where.

· Self-attention: a weighted average.



Transformers - a closer look

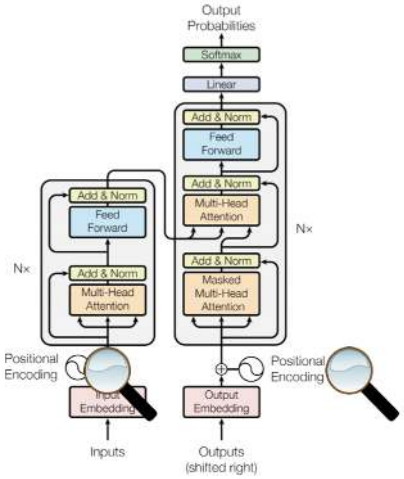
Uses fixed positional encoding based on trigonometric series, in order for the model to make use of the order of the sequence

Positional Encoding

dimension

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$



Self-Attention: complexity

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Where n is the sequence length, d is the representation dimension, k is the convolutional kernel size and r is the size of the neighborhood.

Considering that most sentences have a similar dimension that the representation dimension, self attention is very efficient.

Transformers - training tricks

· ADAM optimizer with proportional learning rate:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

· Residual dropout

· Label smoothing

· Checkpoint averaging

Transformers - Results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Transformers - summary

· Significant improved SOTA in machine translation

· Launched a new deep-learning revolution in MLP

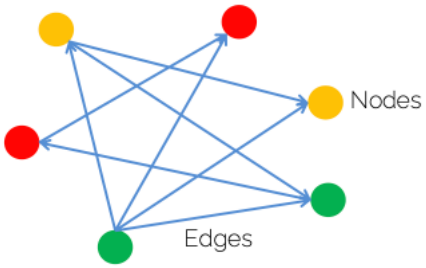
· Building block of NLP models like BERT (Google) or GPT/ChatGPT (OpenAI)

· Bert has been heavily used in Google Search

· And eventually made its way to computer vision (and other related fields)

LECTURE 12: Advanced DL topics

Graph Neural Network



A graph

· Node: a concept
· Edge: a connection between concepts

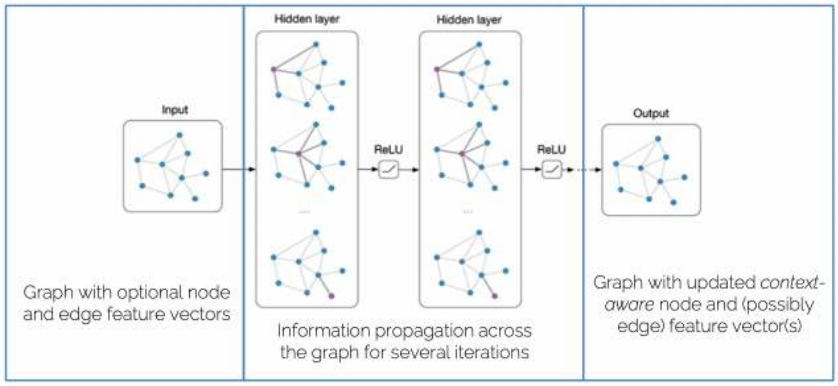
Deep learning on graphs

· Generalization of neural networks that can operate on graph-structured domains:

– Scarselli et al. "The Graph Neural Network Model". IEEE Trans. Neur. Net 2009.
– Kipf et al. "Semi-Supervised Classification with Graph Convolutional Networks. ICLR 2016.
– Gilmer et al. "Neural Message Passing for Quantum Chemistry". ICML 2017
– Battaglia et al. "Relational inductive biases, deep learning, and graph networks". arXiv 2018 (review paper)

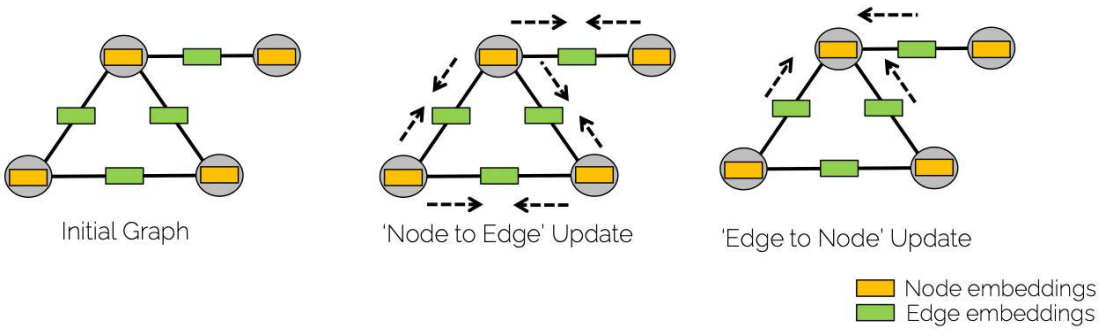
· Key challenges:
- Variable sized inputs (number of nodes and edges)
- Need invariance to node permutations

General Idea



Message Passing Networks

· We can divide the propagation process in two steps: "node to edge" and "edge to node" updates.



Node to edge updates

· At every message passing step l first do:

$$h_{(i,j)}^{(l)} = \mathcal{N}_e\left([h_i^{(l-1)}, h_{(i,j)}^{(l-1)}, h_j^{(l-1)}]\right)$$

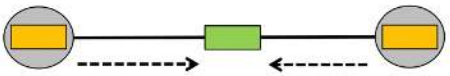
Embedding of node i in the previous message passing step

Embedding of edge (i,j) in the previous message passing step

Embedding of node j in the previous message passing step

$$h_{(i,j)}^{(l)} = \mathcal{N}_e\left([h_i^{(l-1)}, h_{(i,j)}^{(l-1)}, h_j^{(l-1)}]\right)$$

Learnable function (e.g. MLP) with shared weights across the entire graph



Edge to node updates

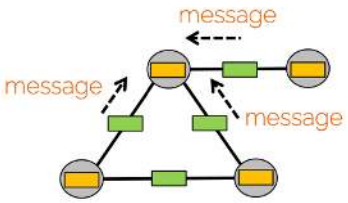
· After a round of edge updates, each edge embedding contains information about its pair of incident nodes.

· Then, edge embedding are used to update nodes:

$$m_i^{(l)} = \Phi\left(\left\{h_{(i,j)}^{(l)}\right\}_{j \in N_i}\right)$$

Order invariant operation (e.g. sum, mean, max)

Neighbors of node i



$$m_i^{(l)} = \Phi\left(\left\{h_{(i,j)}^{(l)}\right\}_{j \in N_i}\right)$$

$$h_i^{(l)} = \mathcal{N}_v\left(\left[m_i^{(l)}, h_i^{(l-1)}\right]\right)$$

Learnable function (e.g. MLP) with shared weights across the entire graph

The aggregation provides each node embedding with contextual information about its neighbors

GNN Application

· Node or edge classification
  - Identifying anomalies such as spam, fraud
  - Relationship discovery for social networks, search networks

· Modeling epidemiology
  - Spatio-temporal graph

· Traffic forecasting

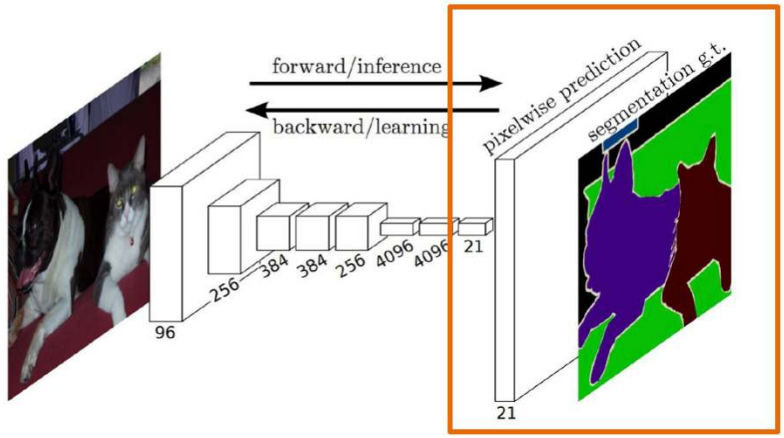
· Scene graph generation

· 3D mesh generation

Generative Models

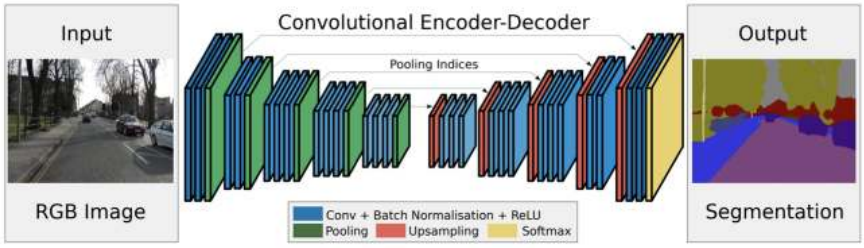
Semantic Segmentation (FCN)

· Recall the Fully Convolutional Networks



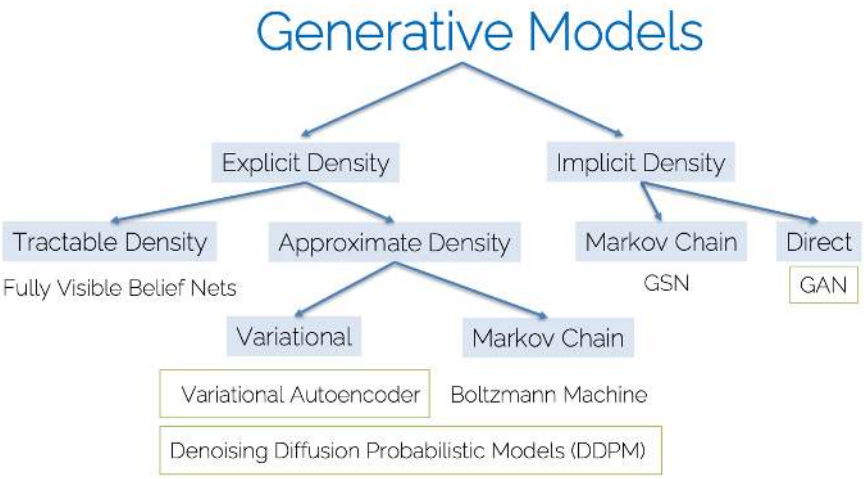
Can we do better?

SegNet



Generative Models

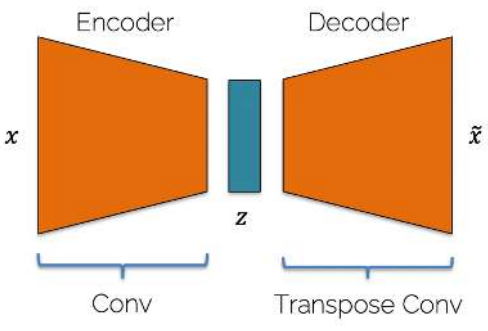
· Given training data, how to generate new samples from the same distribution



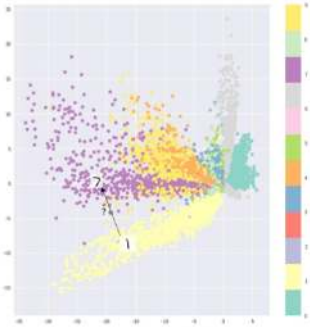
Variational Autoencoders

Autoencoders

· Encode the input into a representation (bottleneck) and reconstruct it with the decoder



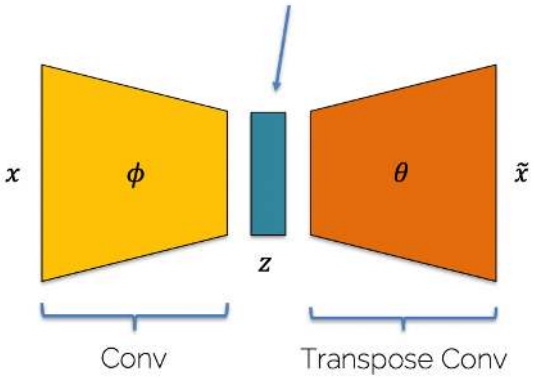
· Encode the input into a representation (bottleneck) and reconstruct it with the decoder



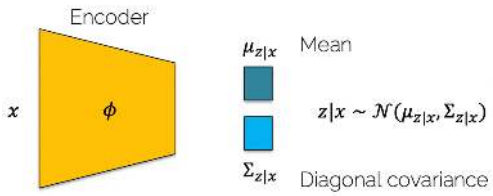
Latent space learned by autoencoder on MNIST

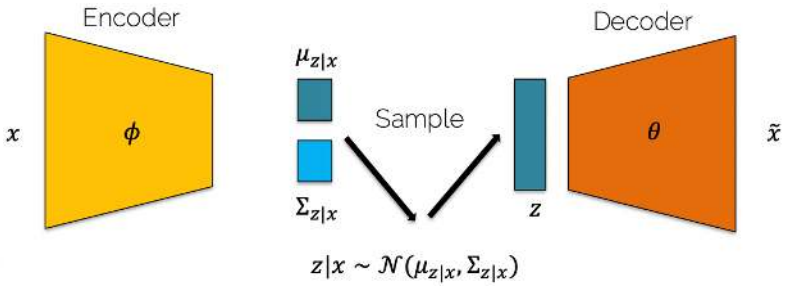
Variational Autoencoder

Goal: Sample from the latent distribution to generate new outputs!



· Latent space is now a distribution
· Specifically it is a Gaussian





· Training: loss makes sure the latent space is close to a Gaussian and the reconstructed output is close to the input.

· Test: Sample from the latent space.

Autoencoder vs Variational Autoencoder

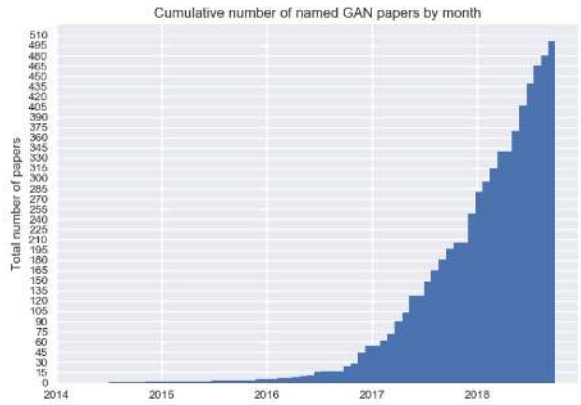
Autoencoder Overview

· Autoencoders (AE)
- Reconstruct input
- Unsupervised learning

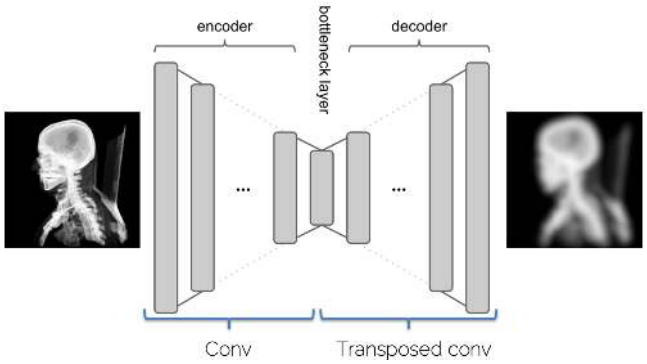
· Variational Autoencoders (VAE)
- Probability distribution in latent space (e.g., Gaussian)
- Interpretable latent space (head pose, smile)
- Sample from model to generate output

Generative Adversarial Networks (GANs)

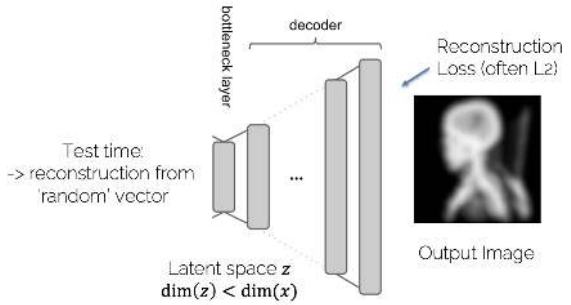
Generative Adversarial Networks (GANs)

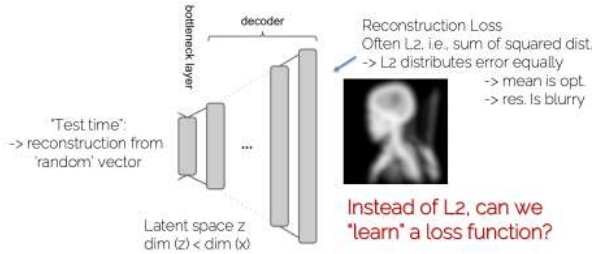


Autoencoder

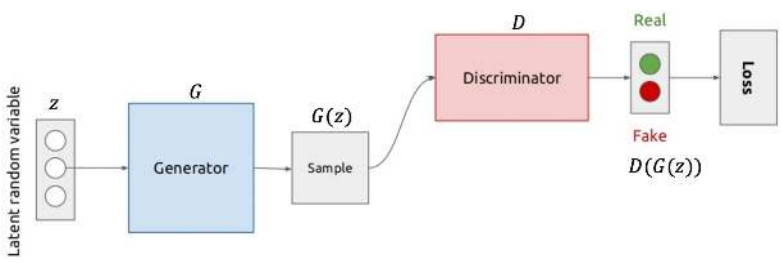


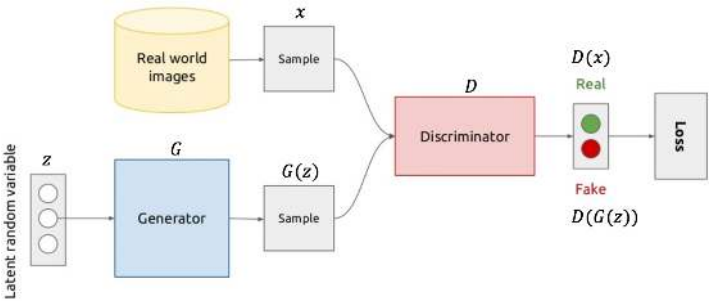
Decoder as Generative Model

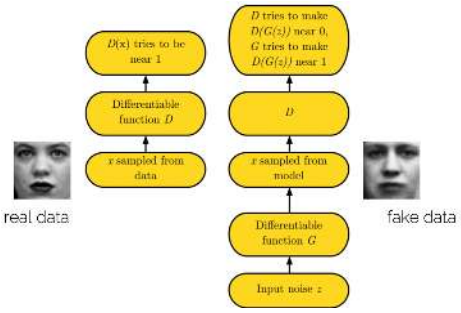




Generative Adversarial Networks (GANs)







GANs: Loss Function

- Discriminator loss

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_{data}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log\left(1 - D(G(\boldsymbol{z}))\right)$$

binary cross entropy

- Generator loss

$$J^{(G)} = -J^{(D)}$$

- Minimax Game:
  - G minimizes probability that D is correct
  - Equilibrium is saddle point of discriminator loss
    - D provides supervision (i.e., gradients) for G

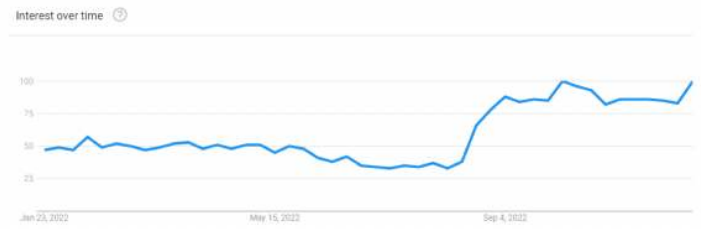
GAN Applications

GAN Application

- BigGAN: HD Image Generation
- StyleGAN: Face Image Generation
- Cycle GAN: Unpaired Image-to-Image Translation
- SPADE: GAN-Based Image Editing
- Texturify: 3D Texture Generation

Diffusion

Diffusion - Search Interest



Diffusion Models

· Class of generative models

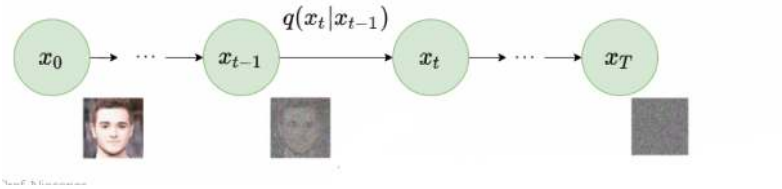
· Achieved state-of-the-art image generation (DALLE-2, Imagen, StableDifussion)

· What is diffusion?

Diffusion Process

· Gradually add noise to input image x_0 in a series of T time steps

· Neural network trained to recover original data



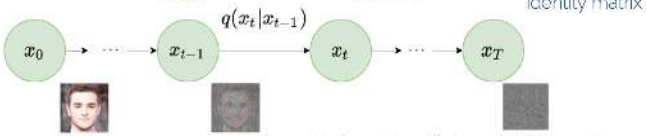
Forward Diffusion

· Markov chain of T steps

- Each step depends only on previous
- Adds noise to x_0 sampled from real distribution $q(x)$

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \boldsymbol{\mu}_t = \sqrt{1-\beta_t}x_{t-1}, \boldsymbol{\Sigma}_t = \beta_t\mathbf{I})$$

mean variance identity matrix



- Go from x_0 to x_T:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

Reparametrization

- Define $\alpha_t = 1 - \beta_t$, $\overline{\alpha_t} = \prod_{s=0}^{t} \alpha_s$, $\epsilon_0, \ldots, \epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

$$\begin{aligned} x_t &= \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_{t-1} \\ &= \sqrt{\alpha_t}x_{t-2} + \sqrt{1-\alpha_t}\epsilon_{t-2} \\ &= \cdots \\ &= \sqrt{\overline{\alpha_t}}x_0 + \sqrt{1-\overline{\alpha_t}}\epsilon_0 \end{aligned}$$

$$x_t \sim q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\overline{\alpha_t}}x_0, (1-\overline{\alpha_t})\mathbf{I})$$

Reverse Diffusion

- $x_{T\to\infty}$ becomes a Gaussian distribution

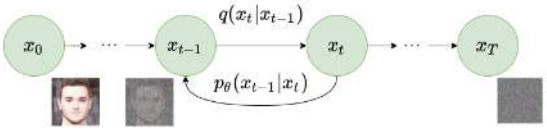
- Reverse distribution $q(x_{t-1}|x_t)$
  - Sample $x_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and run reverse process
  - Generates a novel data point from original distribution

Approximate Reverse Process

- Approximate $q(x_{t-1}|x_t)$ with parameterized model p_θ (e.g., deep network)

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$



Training a Diffusion Model

- Optimize negative log-likelihood of training data

$$L_{VLB} = \mathbb{E}_q[\underbrace{D_{KL}(q(x_T|x_0)||p_\theta(x_T))}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))}_{L_{t-1}} \underbrace{- \log p_\theta(x_0|x_1)}_{L_0}]$$

- Nice derivations: https://lilianweng.github.io/posts/2021-07-11-diffusion-models

- $L_{t-1} = D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))$
- Comparing two Gaussian distributions
- $L_{t-1} \propto \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2$

- Predicts diffusion posterior mean

Diffusion Model Architecture

· Input and outputs dimensions must match

· Highly flexible to architecture design

· Commonly implemented with U-Net architectures

Applications for Diffusion Models

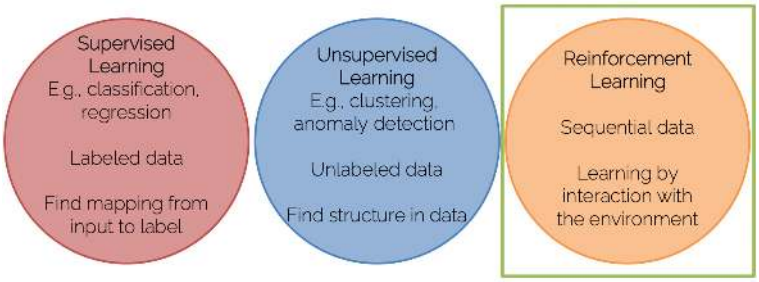
· Text to image

· Image inpainting and outpainting

· Text-to-3D Neural Radiance Fields

Reinforcement Learning

Learning Paradigms in ML



In a Nutshell

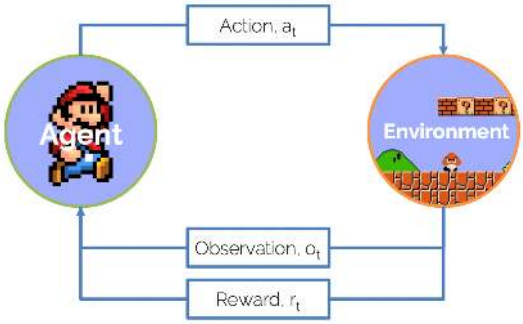
· RL-agent is trained using the “carrot and stick” approach

· Good behavior is encouraged by rewards

· Bad behavior is discouraged by punishment

Agent and Environment

· Reward and next state are functions of current observation Ot and action at only



Characteristic of RL

· Sequential, non i.i.d. data (time matters)

· Action have an effect on the environment —> Change future input

· No supervisor, target is approximated by the reward signal

History and State

· The agent makes decisions based on the history h of observations, actions and rewards up to time-step t

$$h_t = o_1, a_1, r_1, \dots, a_{t-1}, r_{t-1}, o_t$$

· The state s contains all the necessary information from h —< s is a function of h $s_t = f(h_t)$

Markov Assumption

· Problem: History grows linearly over time

· Solution: Markov Assumption

· A state St is Markov if and only if: $\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, \dots s_t]$

· The future is independent of the past given the present

Mathematical Formulation

- The RL problem is a Markov Decision Process (MDP) defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

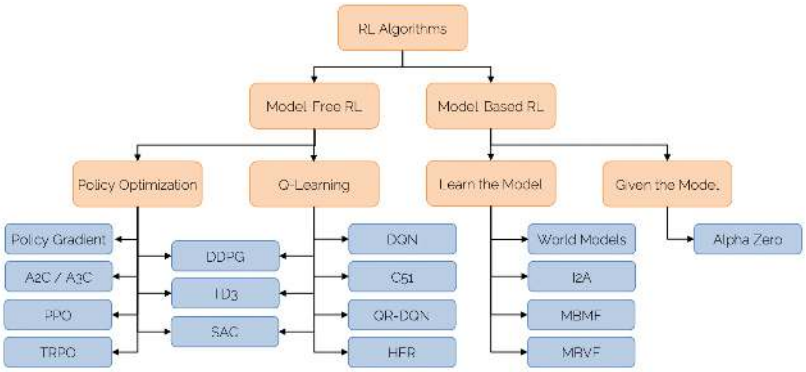
  $\mathcal{S}$: Set of possible states
  $\mathcal{A}$: Set of possible actions
  $\mathcal{R}$: Distribution of reward given (state, action) pair
  $\mathbb{P}$: Transition probability of a (state, action) pair
  γ : Discount factor (discounts future rewards)

Components of an RL Agent

- Policy π : Behavior of the agent
  -> Mapping from state to action: $a = \pi(s)$

- Value-, Q-Function: How good is a state or (state, action) pair
  -> Expected future reward

Taxonomy of RL Algorithm



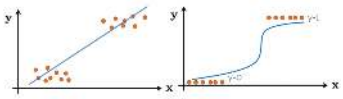
I2DL Overview

Machine Learning Basics

- Unsupervised vs Supervised Learning

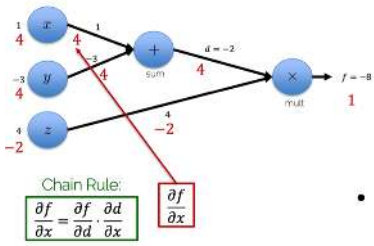
- Data splitting

- Linear vs logistic regression

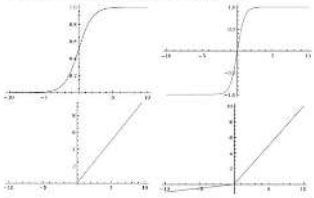


Intro to neural Networks

- Backpropagation



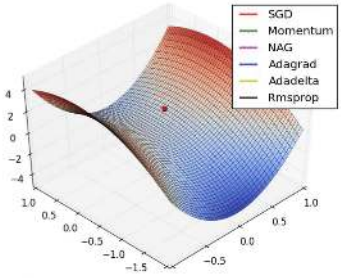
- Activation functions



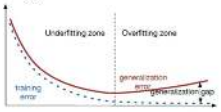
- Loss functions
  - Comparison & effects

Training Neural Networks

- Gradient Descent/ SGD



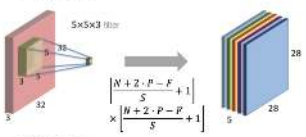
- Regularization



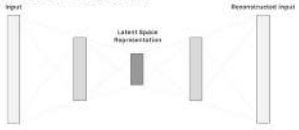
- Parameter & interpre

Typology of Neural Networks

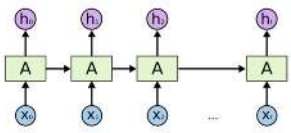
- CNNs



- Autoencoder



- RNNs



- GANs