kptan86@gmail.com

Technical difficulties

- This is about all the theorectical foundation for deep learning.
- The majority of effort in deep learning is spent on
  1. Solving technical difficulties and
  2. Finding applications / use-cases

(Personally, I'd like to see more theorectical research ... , e.g.

https://arxiv.org/abs/1804.04775 (https://arxiv.org/abs/1804.04775)

Will talk more at the end of presentation)

Technical difficulties around neural network

Very notable characteristic of neural network: **Large number of latent variables (weights) to be fitted**

For neural network to work, we need

No	Requirements
1	Large amount of data for the fitting
2	Efficient optimization methods for the fitting
3	Specfial attention on ill-positioning and overfitting prevention

Technical difficulties around neural network

Very notable characteristic of neural network: **Large number of latent variables (weights) to be fitted**

For neural network to work, we need

No	Requirements	Solution
1	Large amount of data for the fitting	Network architecture, Reinforcement learning, other tricks (no clear solution yet)
2	Efficient optimization methods for the fitting	Back-propagation (+architecture/activation function design), hardware solution
3	Specfial attention on ill-conditioning and overfitting prevention	Drop-out regularization, initialization method

DO NOT DISTRIBUTE

kptan86@gmail.com

Fitting weights (and bias)

(remember the functional form)

$$y = F(\ldots \sum_j F(w_j, b_j, (\sum_i F(w_i, b_i, x_i)))\ldots)$$

objective: Finding the best ws and bs that fits the data

Elements for weights opimization

- Objective function
  - Any monotonic function with ground truth as global minimum will do
  - Could consider preventing overfitting by regularization
    - (note: will discuss in more depth later on this)
- Optimizer / Optimization methods
  - Gradient-less methods
    - Nelder-Mead, Powell, Brute-force, Basin-hopping, Simulated annealing, Tabu-Search, **Genetic algorithm** etc
  - Gradient-based methods
    - Conjugate gradient, BFGS, Newton, etc

Backpropagation

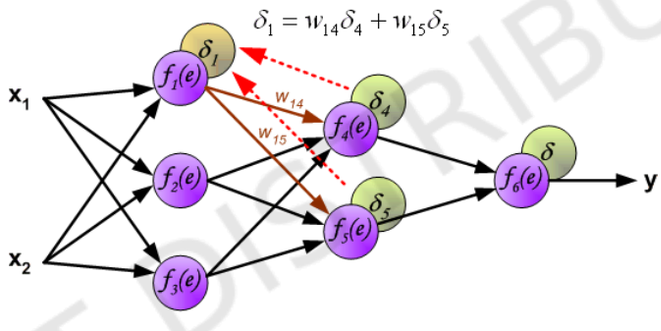
Modern neural-networks typically employs a special form of **gradient-based method** known as "**error backpropagation**"

- Based on message passing framework.
- Efficient computation by exploting the the architecture of the network
- Update weights of a neuron based on the final error "back-propagated" through the computation path (based on a learning rate η)

In summary, weight is updated with $\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} = -\eta \cdot \delta_j o_i$

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} (o_j - t_j)o_j(1-o_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{l \in L} w_{jl}\delta_l)o_j(1-o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

where E is error, η is learning rate, o_i is output of neuron i, L is the connected layer to the neuron



Derivation of backpropagation

see https://en.wikipedia.org/wiki/Backpropagation (https://en.wikipedia.org/wiki/Backpropagation) and/or https://en.wikipedia.org/wiki/Delta_rule (https://en.wikipedia.org/wiki/Delta_rule)

kptan86@gmail.com

Vanishing gradient problem

Rewriting the previous equations .. $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}o_i$

$\frac{\partial E}{\partial o_j} = \sum_{l \in L}\left(\frac{\partial E}{\partial \text{net}_l}\frac{\partial \text{net}_l}{\partial o_j}\right) = \sum_{l \in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial \text{net}_l}w_{jl}\right)$

- L is the next layer (to the direction of output) wrt to J (current layer)
- $\frac{\partial o_j}{\partial \text{net}_j}$ is just the derivatives of the activation function

Rearranging gives

$$\frac{\partial E}{\partial w_{ij}} = o_i \sum_{l \in L}(w_{jl}\frac{\partial E}{\partial o_l}\boxed{\frac{\partial o_l}{\partial \text{net}_l})\frac{\partial o_j}{\partial \text{net}_j}}$$

Vanishing gradient problem

Suppose we use logistic function

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

The derivative is

$f'(x) = e^x(1 + e^x)^{-2} = f(x)(1 - f(x))$

notice that $|f(x)| <= 1$, it follows that $|f'(x)| <= 1$.

Gradient vanishes when propagating to the top layers !

Solution to vanishing gradient problem

1. Use alternative techniques other than backpropagation
2. Use other activation functions
3. Break deep network into blocks of shallow networks

Solution to vanishing gradient problem

Use alternative technique than backpropagation

e.g. restrictive Boltzmann machine, deep-belief network (training layer-by-layer) etc.

1. J. Schmidhuber., "Learning complex, extended sequences using the principle of history compression," Neural Computation, 4, pp. 234–242, 1992.
2. Hinton G (2009). "Deep belief networks". Scholarpedia. 4 (5): 5947. doi:10.4249/scholarpedia.5947.

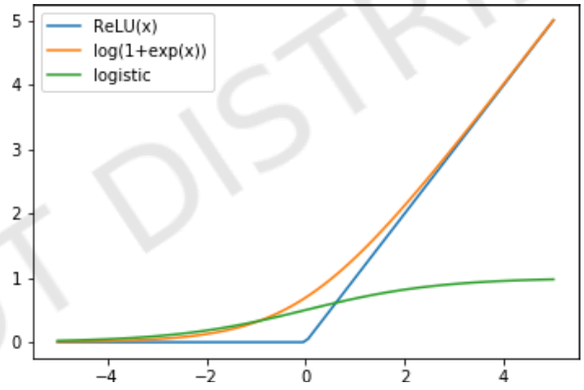
(*note: not so popular now)

kptan86@gmail.com

Solution to vanishing gradient problem

Use other activation function

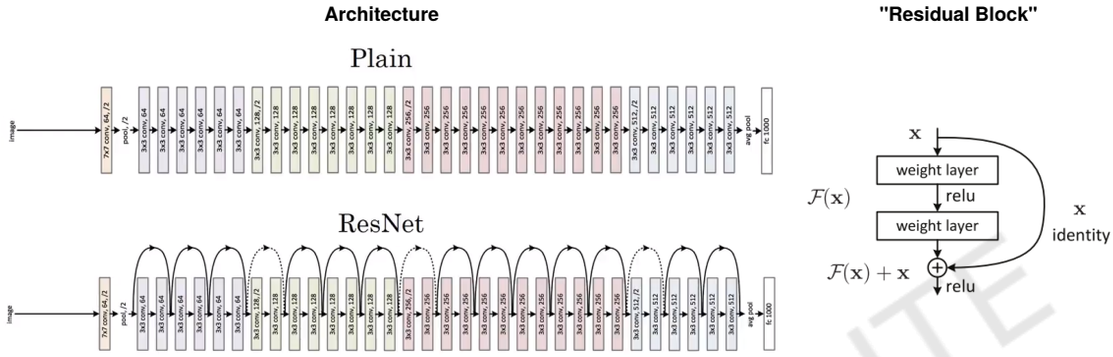
$\mathrm{ReLU}(x) = \max(0, x)$ being the most popular one (notice: the derivative is constant)



Solution to vanishing gradient problem

Break deep network into blocks of shallow networks

e.g. ResNet and Inception net



Vanishing variance problem

Let us look at the other part of the equation ...

$$\frac{\partial E}{\partial w_{ij}} = o_i \boxed{\sum_{l \in L} (w_{jl} \frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial \mathrm{net}_l})} \frac{\partial o_j}{\partial \mathrm{net}_j}$$

kptan86@gmail.com

Vanishing variance problem

It turns out that the variance (roughly equivalent to the range of value) of the weight update is related to the number of connections [Glorot and Bengio, ICIS 2010].

The relation (under certain assumptions) is given by the following form*

$$\mathrm{Var}\left(\frac{\partial E}{\partial w}\right) = \boxed{[n\mathrm{Var}(W)]^d}\mathrm{Var}(x)\mathrm{Var}(\frac{\partial E}{\partial s})$$

where W is the weights, n is the number of neuron in the layer, d is the number of subsequent layers, E is the error, s is the weighed sum.

(*note the equation is exact only when all layers have the same number of neurons)

The variance could vanish (or explode) after many layers !

Vanishing variance problem

$$\mathrm{Var}\left(\frac{\partial E}{\partial w}\right) = \boxed{[n\mathrm{Var}(W)]^d}\mathrm{Var}(x)\mathrm{Var}(\frac{\partial E}{\partial s})$$

Solution

To alleviate the problem, it should be rectified such that

$$n\mathrm{Var}(W) \approx 1$$

As number of the connections (n) are pre-determined and weights (W) are fitted during optimzation, there is not much room for control, except at **initialization**.

Recall that for uniform distribution $U(a, b)$, the variance is $(b-a)^2/12$, it follows that for a **zero-mean, symmetry uniform initialization scheme**, the values should be draw from the following distribution

$$W = U\left(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right)$$

Technical difficulties around neural network

Very notable characteristic of neural network: **Large number of latent variables (weights) to be fitted**

For neural network to work, we need

No	Requirements	Solution
1	Large amount of data for the fitting	Network architecture, Reinforcement learning, other tricks (no clear solution yet)
2	Efficient optimization methods for the fitting	Back-propagation (+architecture/activation function design), hardware solution
3	Specfial attention on ill-conditioning and overfitting prevention	Drop-out regularization, initialization method

DO NOT DISTRIBUTE

kptan86@gmail.com

ill-positioning improvement and overfitting prevention

ill-positioning

Mathematically, a well-posed problem should satisfy:

- Existence of a solution
- Uniquenes of solution
- Smoothness of solution w.r.t. with the initial conditions changes

Overfitting

- Overfitting usually occurs when there are more parameters in a model than can be justified by the data.

With large number of latent variables, neural network is very prone to ill-conditioning (esp. condition 2) and overfitting !

ill-positioning improvement and overfitting prevention

- ill-positioning problem arises as there are many different solutions fit the data equally well
- Soln: among all candidate solutions, choose one.
  - The chosen solution should have some property "better" than the other competing solutions
  - (The *property* is to be defined arbitrarily (i.e. it is a "prior"))

This formal choice for solution is called **Regularization**.

To implement, usually a "regularization term" $R(.)$ is added term to the loss function $L(.)$:

$$\min_f \sum_{i=1}^{n} L(f(x_i), y_i) + \lambda R(f)$$

where f is the model (with all the parameters), λ is a hyperparameter controlling the importance of regularization term.

- $R(.)$ takes the model parameters as input, and compute the specified *property*.
- **In principle,** $R(.)$ **could be arbitrary**.

DO NOT DISTRIBUTE

kptan86@gmail.com

L2 regularization

One particularly popular choice for regularization term $R(.)$ is L2 norm.

$$R(f) = ||f||_H^2$$

which corresponds to the *property* that "**model parameters are normally distributed**".

This might not be suitable for neural networks though

- There is no theorectical basis that network latent variables should be normally distributed
  - (not sure if anyone examines this before ?)

Notes on overfitting:

- For many functional forms (e.g. linear models) L2 norm also (roughly) corresponds to the "complexity" of a model, so the regularization also alleviates overfitting problem.
- This is *not* guaranteed for all functional forms
- **Overfitting problem is usually not fully resolved with regularization**
  - Other techniques such as model comparison (AIC, BIC), cross-validation, early stopping, pruning should be additionally used)

L2 regularization derivation

L2 regularization is equivalent of setting a Gaussian prior on the model parameters. To demonstrate this, without loss the generality, let's suppose y is linearly related to x via w, and with data collected as $(x_1, y_1), \ldots, (x_N, y_N)$ with random error of Gaussian form $\epsilon \sim N(\theta, \sigma^2)$. We have the following:

$$y_i = wx_i + \epsilon$$

The likelihood is then of Gaussian form:

$$P = \prod_{n=1}^{N} N(y_n | wx_n, \sigma^2)$$

Let us impose w to be of Gaussian form $w \sim N(0, \sigma_w^2)$, and modify the likelihood with a prior:

$$P = \prod_{n=1}^{N} N(y_n | wx_n, \sigma^2) N(w | 0, \sigma_w^2)$$

Remembering Gaussian function is of the form:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

The log-likelihood can be written as:

$$\log P = \sum_{n=1}^{N} -\frac{1}{\sigma^2}(y_n - wx_n)^2 - \frac{1}{\sigma^2}w^2 + \text{const}$$

It is often to use precision measure λ (reciprocal of variance) for representation, which gives us

$$\log P = \sum_{n=1}^{N} -\frac{1}{\sigma^2}(y_n - wx_n)^2 - \lambda w^2 + \text{const}$$

kptan86@gmail.com

ill-positioning improvement and overfitting prevention

Dropout regularization

An interesting technique used in neural network is "dropout", where randomly selected neurons (and/or connections, depending on the implementation) are "dropped out" or "kept back" during training.

- The operation encourages weights update from backpropagation to be *uniformly distributed among neurons* (rather than concentrating information in diffent neurons).
- This *roughly* corresponds to the *property* that "the homogeneity of weights in the model should be maximized".
- (should also corresponds to **principle of maximum entropy**)
  - (not sure if examined before)

(There could be more efficient and theorectically sound method for regularization to be developed)

DO NOT DISTRIBUTE