Computer Vision:
Training neural networks

Siheng Chen 陈思衡

Training neural networks

Stochastic gradient descent

Dropout

Initialization

Hyperparameter tuning

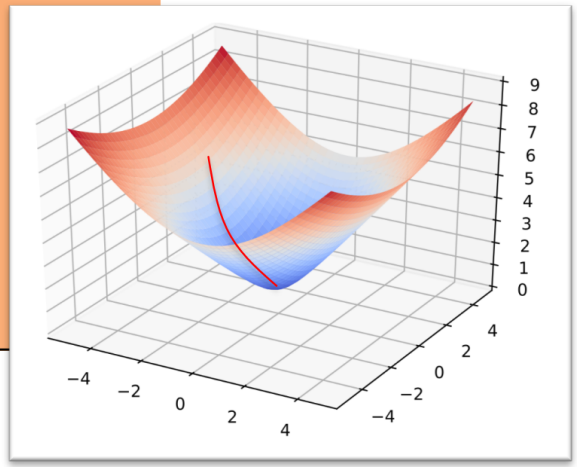
Batch normalization

Stochastic gradient descent

Gradient descent

Algorithm 1 Gradient Descent

1: **procedure** GD($\mathcal{D}$, $\boldsymbol{\theta}^{(0)}$)
2: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(0)}$
3: $\quad$**while** not converged **do**
4: $\quad\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
5: $\quad$**return** $\boldsymbol{\theta}$



original objective:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} J^{(i)}(\boldsymbol{\theta})$$

Stochastic gradient descent

Gradient descent

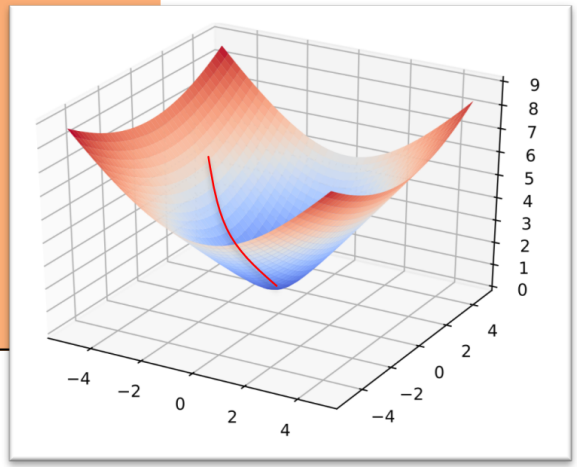
14,197,122 images

Stochastic gradient descent

Gradient descent

Algorithm 1 Gradient Descent

1: **procedure** GD($\mathcal{D}$, $\boldsymbol{\theta}^{(0)}$)
2: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(0)}$
3: $\quad$**while** not converged **do**
4: $\quad\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
5: $\quad$**return** $\boldsymbol{\theta}$



original objective:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} J^{(i)}(\boldsymbol{\theta})$$

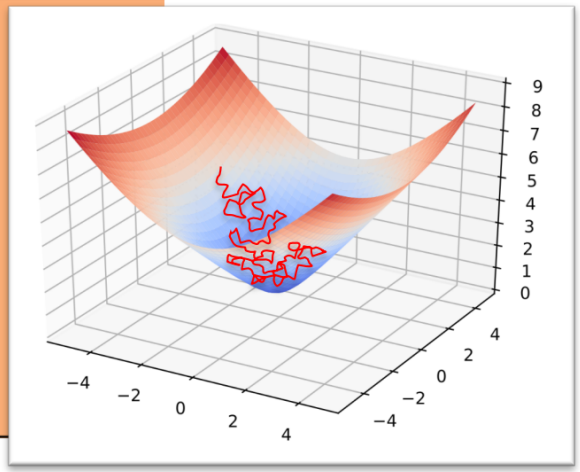
Need to enumerate all the training samples every iteration. Expensive!

Stochastic gradient descent

Stochastic gradient descent

Algorithm 2 Stochastic Gradient Descent (SGD)

1: **procedure** SGD($\mathcal{D}, \boldsymbol{\theta}^{(0)}$)
2: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(0)}$
3: $\quad$ **while** not converged **do**
4: $\quad\quad i \sim \text{Uniform}(\{1, 2, \ldots, N\})$
5: $\quad\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J^{(i)}(\boldsymbol{\theta})$
6: $\quad$ **return** $\boldsymbol{\theta}$



per-example objective:

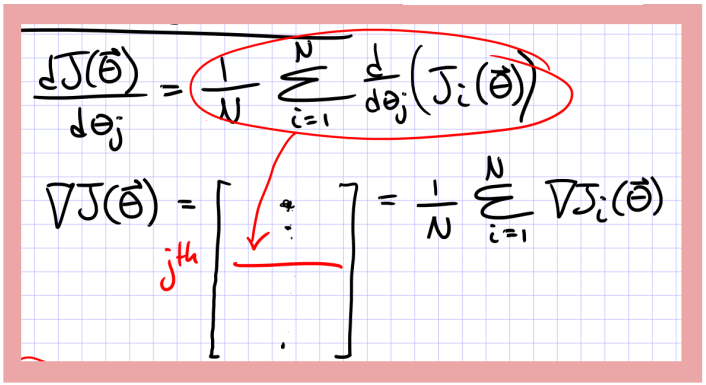
$$J^{(i)}(\boldsymbol{\theta})$$

original objective:

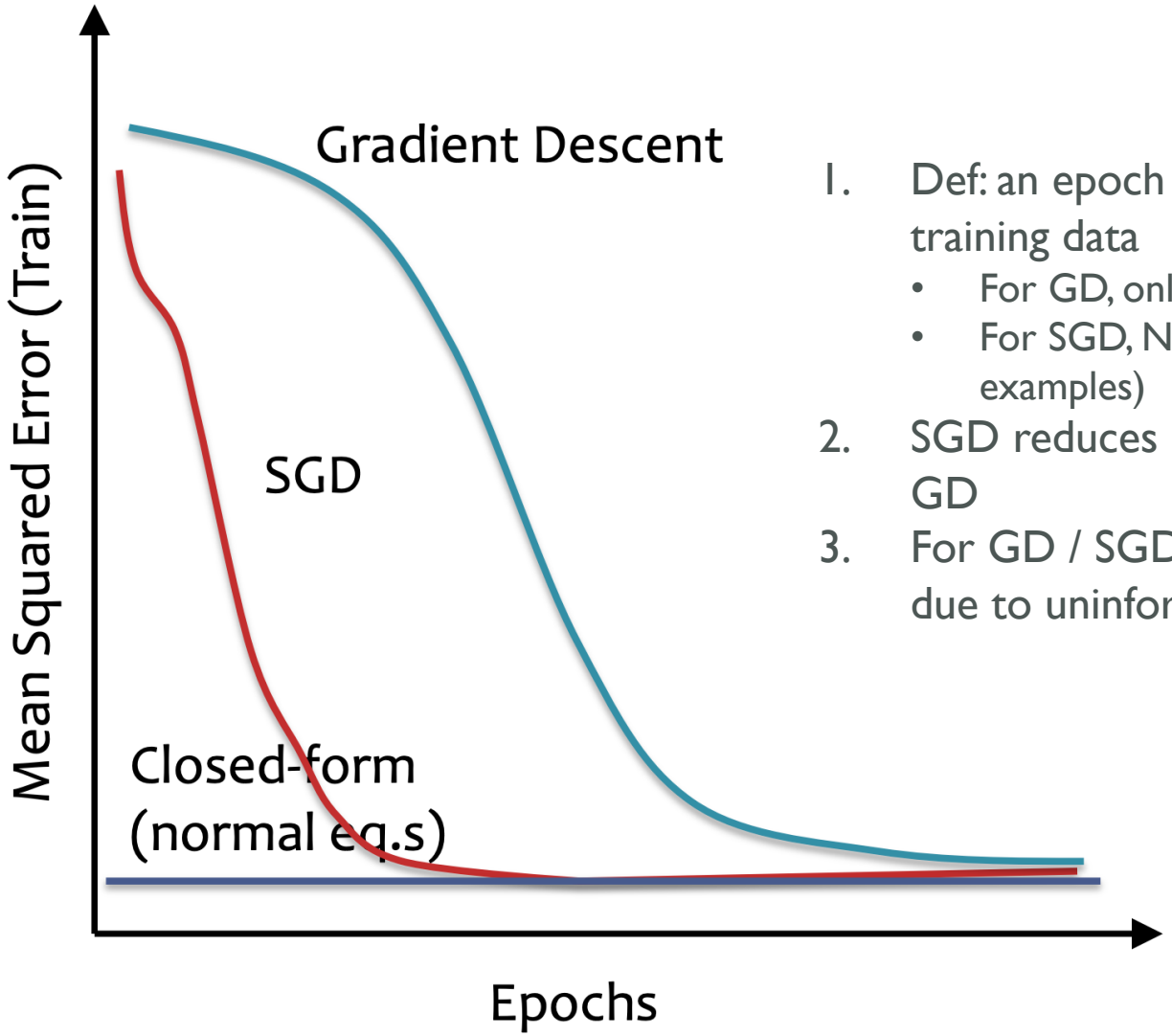
$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} J^{(i)}(\boldsymbol{\theta})$$

Stochastic gradient descent

Stochastic gradient descent



Stochastic gradient descent



1. Def: an epoch is a single pass through the training data
   - For GD, only one update per epoch
   - For SGD, N updates per epoch N = (# train examples)
2. SGD reduces MSE much more rapidly than GD
3. For GD / SGD, training MSE is initially large due to uninformed initialization

Stochastic gradient descent

Batch gradient descent

computes the gradient of the cost function to the parameters for the entire training dataset

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Stochastic gradient descent

computes the gradient of the cost function to the parameters for each training example

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Mini-batch gradient descent

computes the gradient of the cost function to the parameters for every mini-batch of ***n*** training examples

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

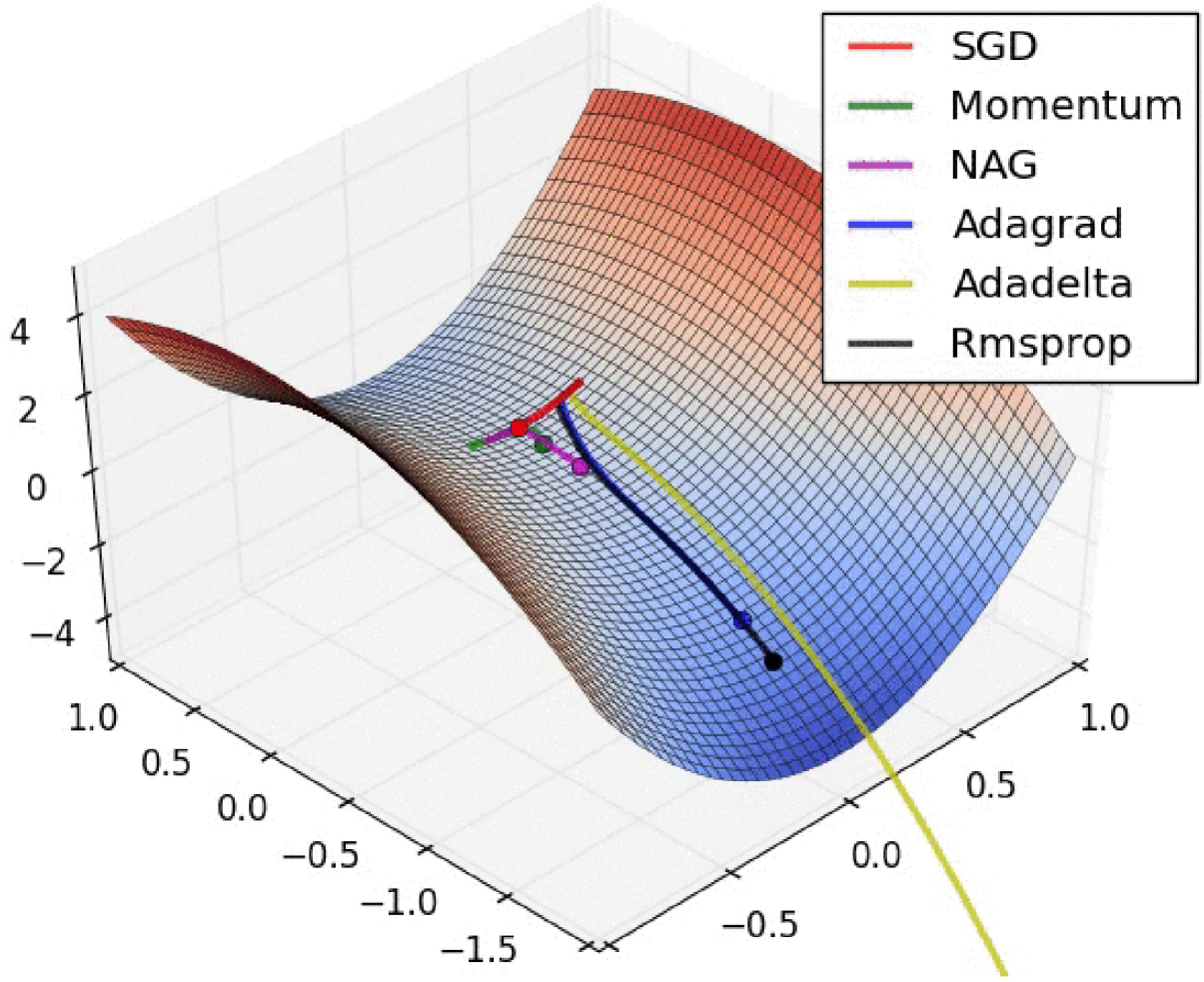
https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

- Local optimum?
- How to choose a proper learning rate?
- Same learning rate for all parameter updates?

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent



Local optimum is not a major concern, but there are many saddle points

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

An overview of gradient descent optimization algorithms*

Sebastian Ruder
Insight Centre for Data Analytics, NUI Galway
Aylien Ltd., Dublin
ruder.sebastian@gmail.com

Abstract

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This article aims to provide the reader with intuitions with regard to the behaviour of different algorithms that will allow her to put them to use. In the course of this overview, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms, review architectures in a parallel and distributed setting, and investigate additional strategies for optimizing gradient descent.

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Momentum

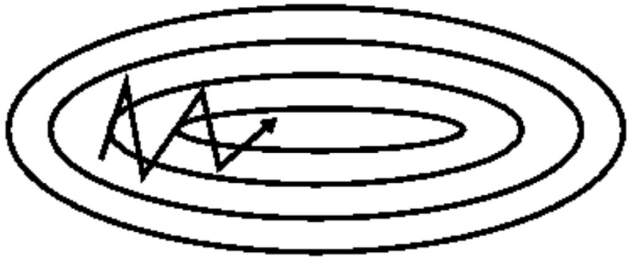
accelerate SGD in the relevant direction and dampen oscillations

0.9

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

(a) SGD without momentum



(b) SGD with momentum

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Nesterov accelerated gradient (NAG)

an approximation of the next position of the parameters

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

look ahead by calculating the gradient not w.r.t. to our current parameters but w.r.t. the approximate future position of our parameters

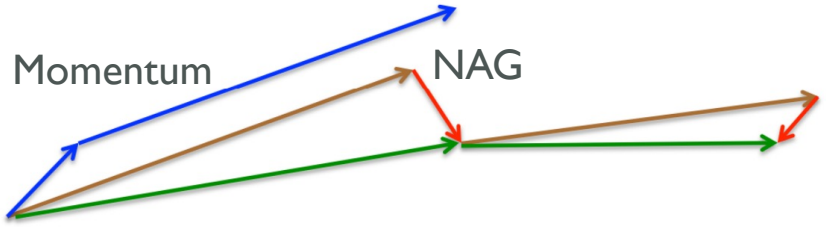


Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Adagrad

adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

element-wise $$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

each diagonal element (i, i) is the sum of the squares of the gradients up to time step t

vectorize $$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Adaptive Moment Estimation (Adam)

computes adaptive learning rates for each parameter

first moment (the mean) 0.9

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

second moment (the uncentered variance) 0.999

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

bias-correction

$$\begin{cases} \hat{m}_t = \dfrac{m_t}{1-\beta_1^t} \\ \hat{v}_t = \dfrac{v_t}{1-\beta_2^t} \end{cases}$$

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

Adam might be the best overall choice

https://arxiv.org/pdf/1609.04747.pdf

Stochastic gradient descent

Additional strategies for optimizing SGD

- Shuffling:

  shuffle the training data after every epoch to avoid bias

- Curriculum learning

  solve progressively harder problems

- Early stopping

  monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough

- Gradient noise:

  adding this noise makes networks more robust to poor initialization and helps training particularly deep and complex networks

https://arxiv.org/pdf/1609.04747.pdf

Training neural networks

Stochastic gradient descent

Dropout

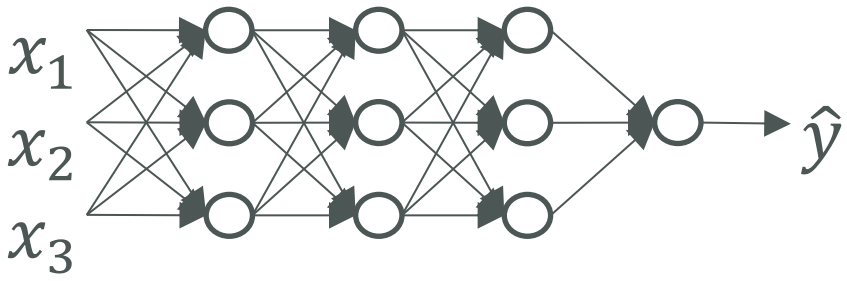
Initialization

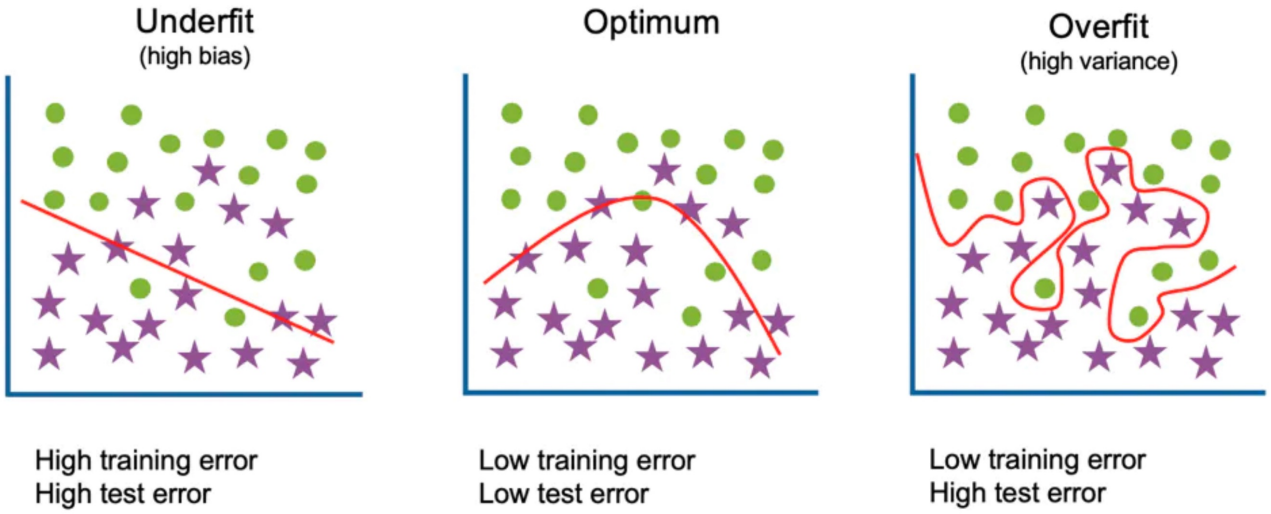
Hyperparameter tuning

Batch normalization

Dropout regularization

The term "dropout" refers to dropping out units in a neural network





https://www.ibm.com/cloud/learn/overfitting

Dropout regularization

Assumption:

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data.

Limit the power of the model parameters

https://arxiv.org/pdf/1609.04747.pdf

Dropout regularization

Regularization

Given objective function: J(θ)

Goal is to find: $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta}) + \lambda r(\boldsymbol{\theta})$

tradeoff between fitting the data and keeping the model simple (*push some parameters smaller*)

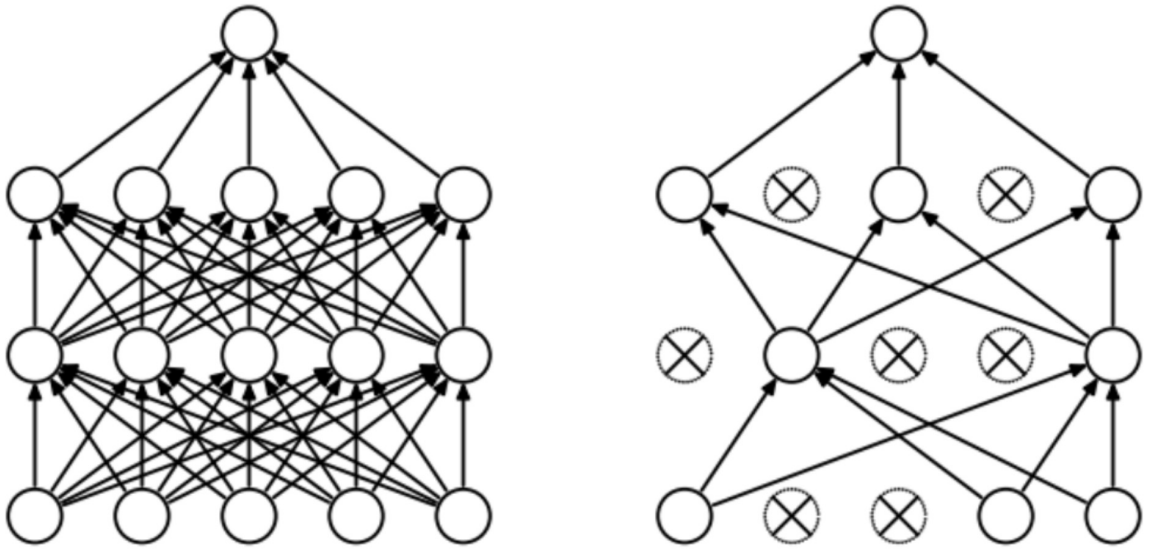
q	$r(\boldsymbol{\theta})$	**yields parameters that are...**	**name**	**optimization notes**
0	$\|\|\boldsymbol{\theta}\|\|_0 = \sum \mathbb{1}(\theta_m \neq 0)$	zero values	L0 reg.	no good computational solutions
1	$\|\|\boldsymbol{\theta}\|\|_1 = \sum \|\theta_m\|$	zero values	L1 reg.	subdifferentiable
2	$(\|\|\boldsymbol{\theta}\|\|_2)^2 = \sum \theta_m^2$	small values	L2 reg.	differentiable

https://arxiv.org/pdf/1609.04747.pdf

Dropout regularization

The term "dropout" refers to dropping out units in a neural network

Intuition: cannot rely on any one feature, so have to spread out weights



(a) Standard Neural Net

(b) After applying dropout.

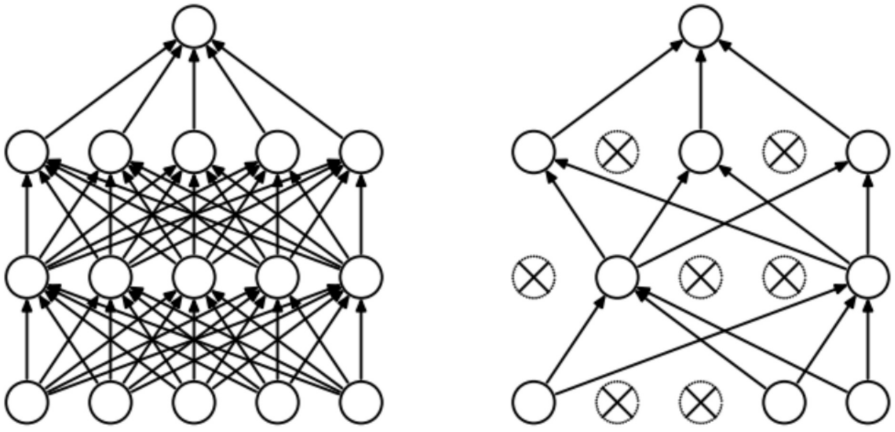
https://arxiv.org/pdf/1609.04747.pdf

Dropout regularization

The term "dropout" refers to dropping out units in a neural network

Training Phase:
For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, *p*, of nodes (and corresponding activations).

Testing Phase:
Use all activations, but reduce them by a factor *p* (to account for the missing activations during training).



(a) Standard Neural Net (b) After applying dropout.

https://arxiv.org/pdf/1609.04747.pdf

Dropout regularization

The term "dropout" refers to dropping out units in a neural network

- Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Dropout roughly doubles the number of iterations required to converge. However, training time for each epoch is less.
- With H hidden units, each of which can be dropped, we have 2^H possible models. In testing phase, the entire network is considered and each activation is reduced by a factor p.

Apply only when you face overfitting issue (lack data)

https://arxiv.org/pdf/1609.04747.pdf

Training neural networks

Stochastic gradient descent

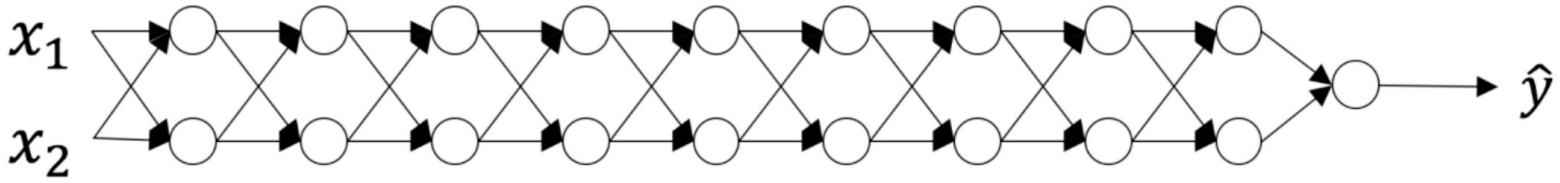
Dropout

Initialization

Hyperparameter tuning

Batch normalization

Weight initialization



$$\hat{y} = a^{[L]} = W^{[L]}W^{[L-1]}W^{[L-2]}\dots W^{[3]}W^{[2]}W^{[1]}x$$

Case 1: A too-large initialization leads to exploding gradients

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

Case 2: A too-small initialization leads to vanishing gradients

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

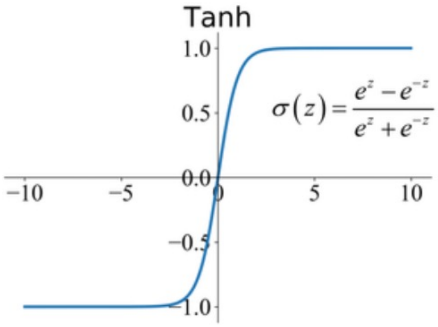
https://www.deeplearning.ai/ai-notes/initialization/

Weight initialization

Key assumptions:

1. Mean of the activations should be zero
2. Variance of the activations should stay the same across layers

$$z^{[l]} = W^{[l]} a^{[l-1]}$$
$$a^{[l]} = tanh(z^{[l]})$$



Find right way to initialize ***W***, such that :

$$Var(a^{[l-1]}) = Var(a^{[l]})$$

https://www.deeplearning.ai/ai-notes/initialization/

Weight initialization

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = tanh(z^{[l]})$$

Additional assumptions:

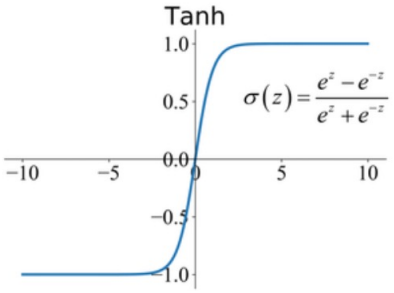
1. Early on in the training, we are in the *linear regime* of tanh $tanh(z^{[l]}) \approx z^{[l]}$
2. Weights are independent and identically distributed

$$Var(w_{kj}^{[l]}) = Var(w_{11}^{[l]}) = Var(w_{12}^{[l]}) = \cdots = Var(W^{[l]})$$

3. Inputs are independent and identically distributed

$$Var(a_j^{[l-1]}) = Var(a_1^{[l-1]}) = Var(a_2^{[l-1]}) = \cdots = Var(a^{[l-1]})$$

4. Weights and inputs are mutually independent



Vectorize: $Var(a^{[l]}) = Var(z^{[l]})$

Element-wise: $Var(a_k^{[l]}) = Var(z_k^{[l]}) = Var(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]}a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]}a_j^{[l-1]})$

$$Var(XY) = E[X]^2Var(Y) + Var(X)E[Y]^2 + Var(X)Var(Y)$$

$$Var(w_{kj}^{[l]}a_j^{[l-1]}) = E[w_{kj}^{[l]}]^2Var(a_j^{[l-1]}) + Var(w_{kj}^{[l]})E[a_j^{[l-1]}]^2 + Var(w_{kj}^{[l]})Var(a_j^{[l-1]})$$

$$Var(z_k^{[l]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]})Var(a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(W^{[l]})Var(a^{[l-1]}) = n^{[l-1]}Var(W^{[l]})Var(a^{[l-1]})$$

Weight initialization

From previous slide, we end up with

$$Var(a^{[l]}) = n^{[l-1]} Var(W^{[l]}) Var(a^{[l-1]})$$

$$n^{[l-1]} Var(W^{[l]}) \begin{cases} < 1 & \Longrightarrow \text{ Vanishing Signal} \\ = 1 & \Longrightarrow Var(a^{[L]}) = Var(x) \\ > 1 & \Longrightarrow \text{ Exploding Signal} \end{cases}$$

To make $Var(a^{[l-1]}) = Var(a^{[l]})$, we need to set

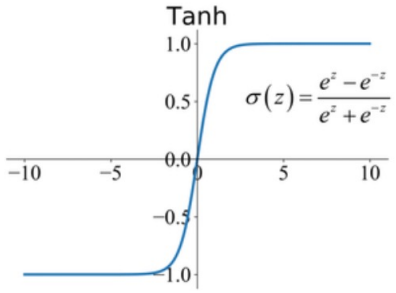
$$Var(W^{[l]}) = \frac{1}{n^{[l-1]}} \qquad \mathcal{N}(0, \frac{1}{n^{[l-1]}})$$

Xavier initialization

https://www.deeplearning.ai/ai-notes/initialization/

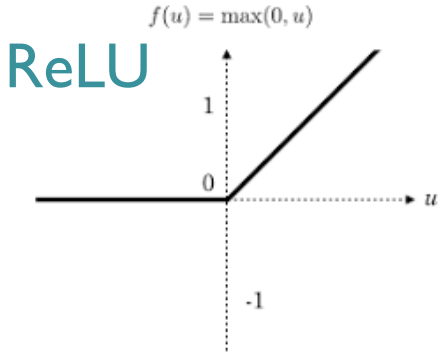
Weight initialization

Xavier initialization



$$\mathcal{N}(0, \frac{1}{n^{[l-1]}})$$

```
W = tf.Variable(np.random.randn(node_in, node_out)) / np.sqrt(node_in)
```



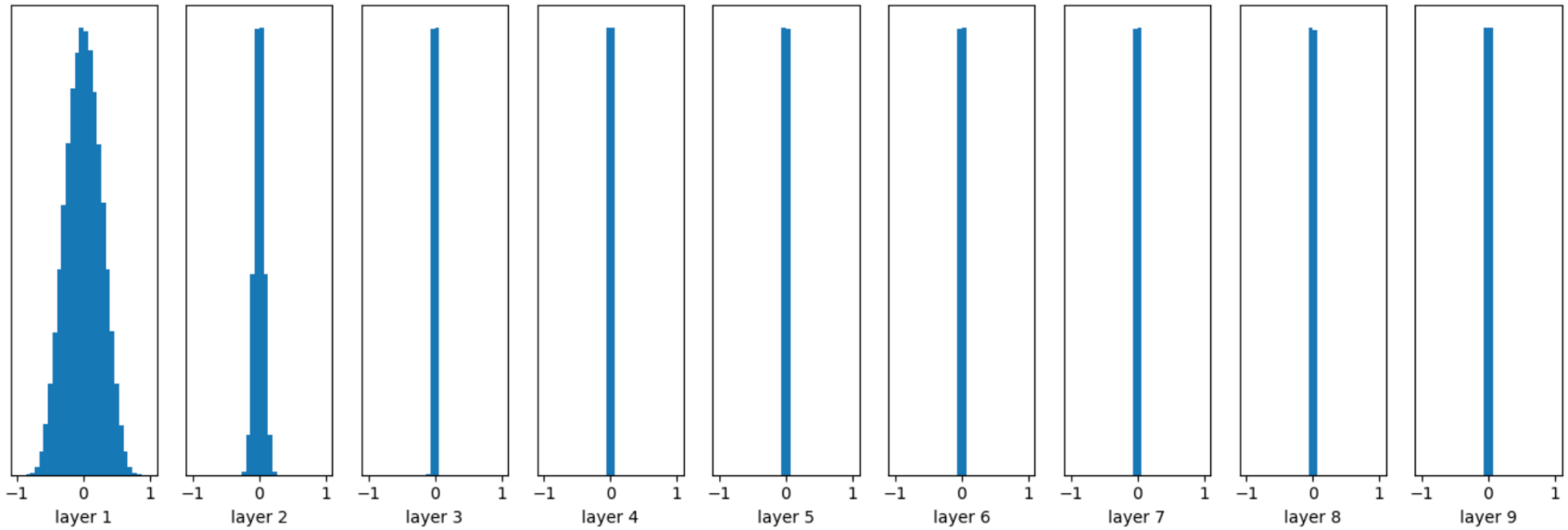
Only half of nodes will be activated, so use $n^{[l-1]}/2$

```
W = tf.Variable(np.random.randn(node_in, node_out)) / np.sqrt(node_in/2)
```

https://www.deeplearning.ai/ai-notes/initialization/

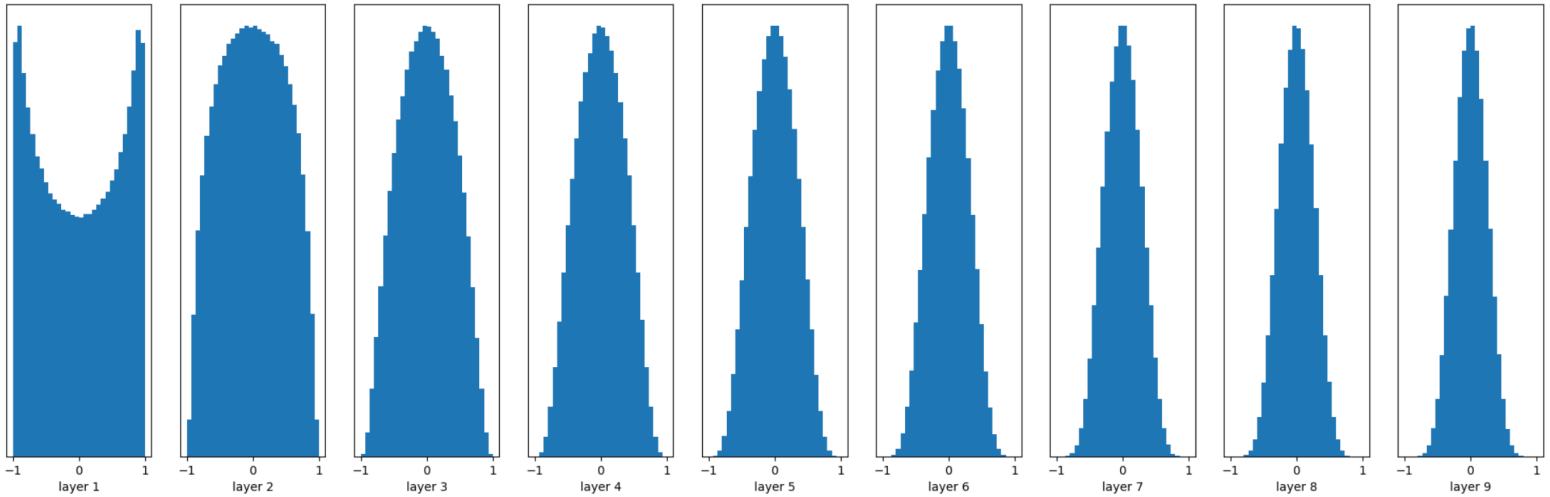
Weight initialization

W = tf.Variable(np.random.randn(node_in, node_out)) * **0.01**



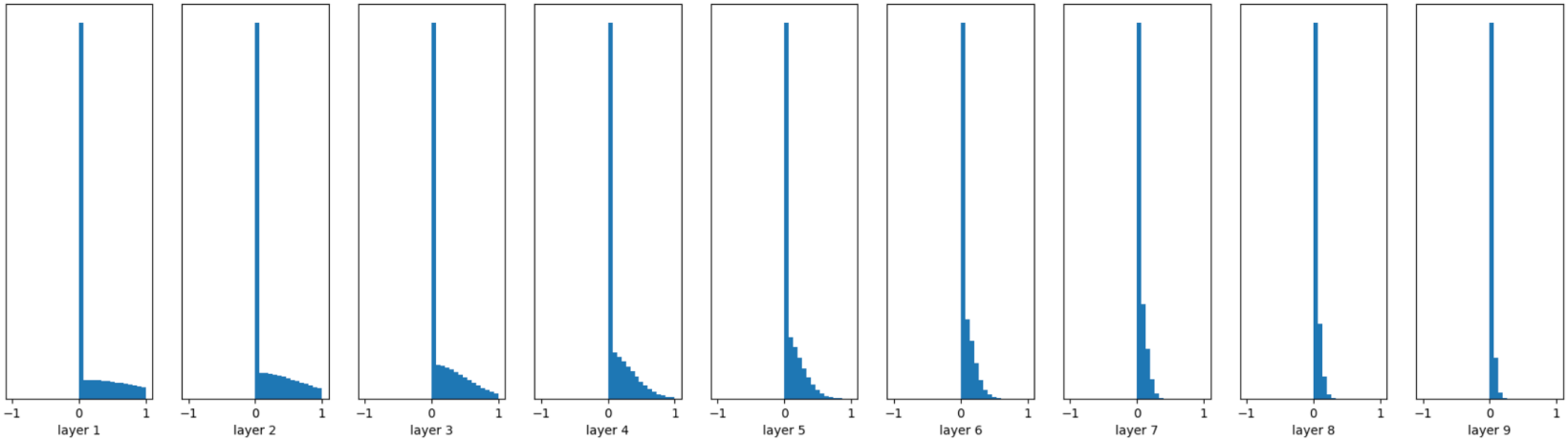
Weight initialization

W **=** tf**.**Variable(np**.**random**.**randn(node_in, node_out)) **/ np.sqrt(node_in)**



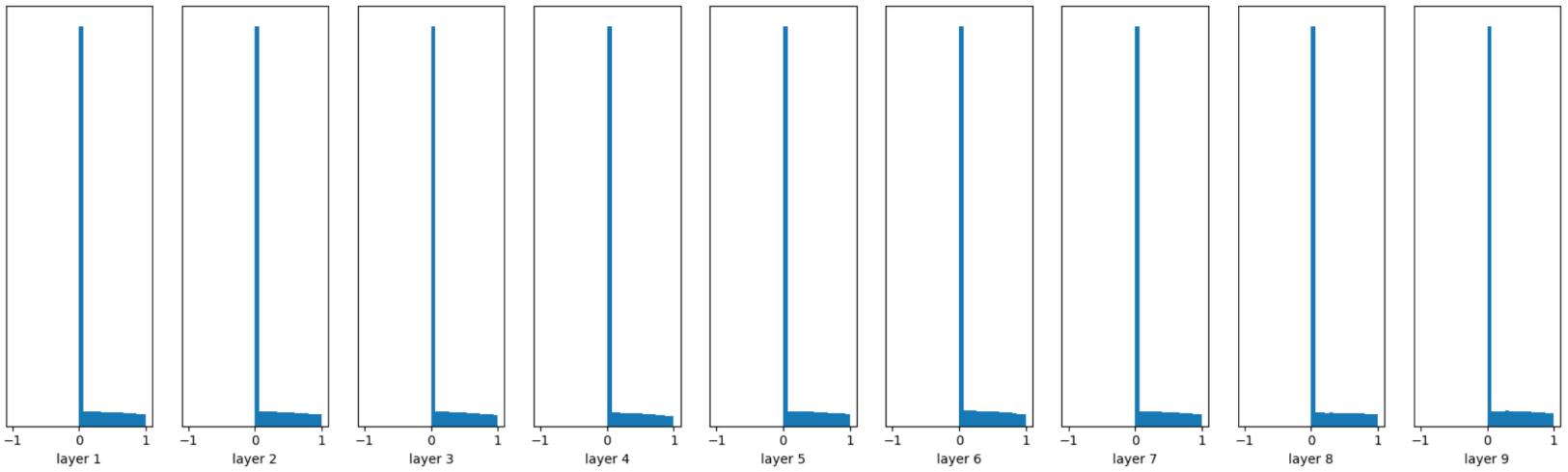
Weight initialization

```
W = tf.Variable(np.random.randn(node_in, node_out)) / np.sqrt(node_in)
.....
fc = tf.nn.relu(fc)
```



Weight initialization

```
W = tf.Variable(np.random.randn(node_in, node_out)) / np.sqrt(node_in / 2)
.....
fc = tf.nn.relu(fc)
```



Training neural networks

Stochastic gradient descent

Dropout

Initialization

Hyperparameter tuning

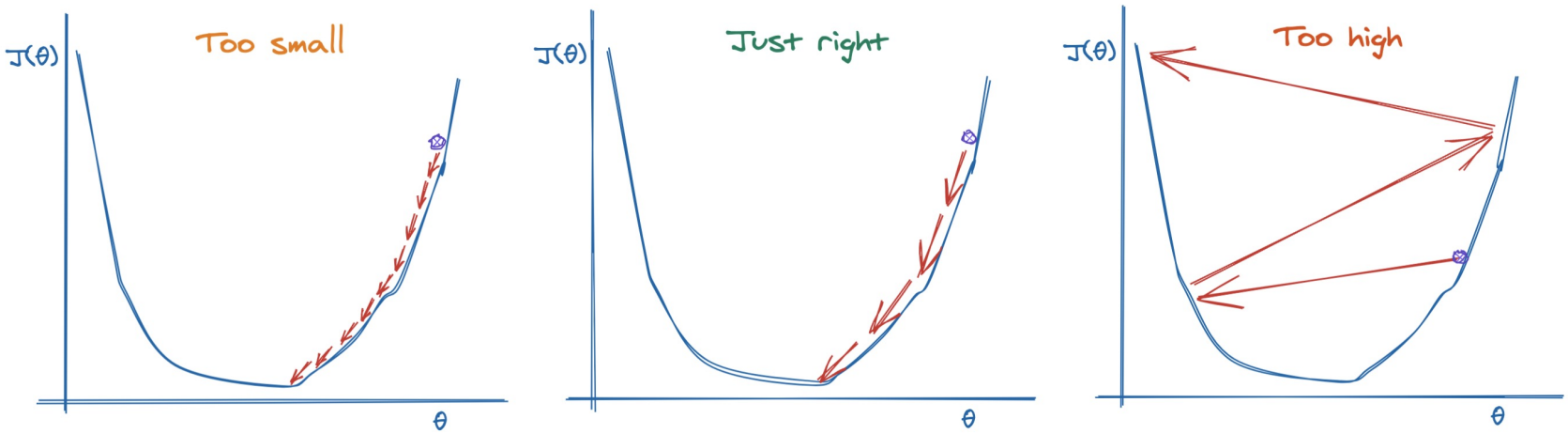
Batch normalization

Hyperparameter tuning

Learning rate, dropout probability, number of layers, neurons per layer…

Hyperparameter tuning

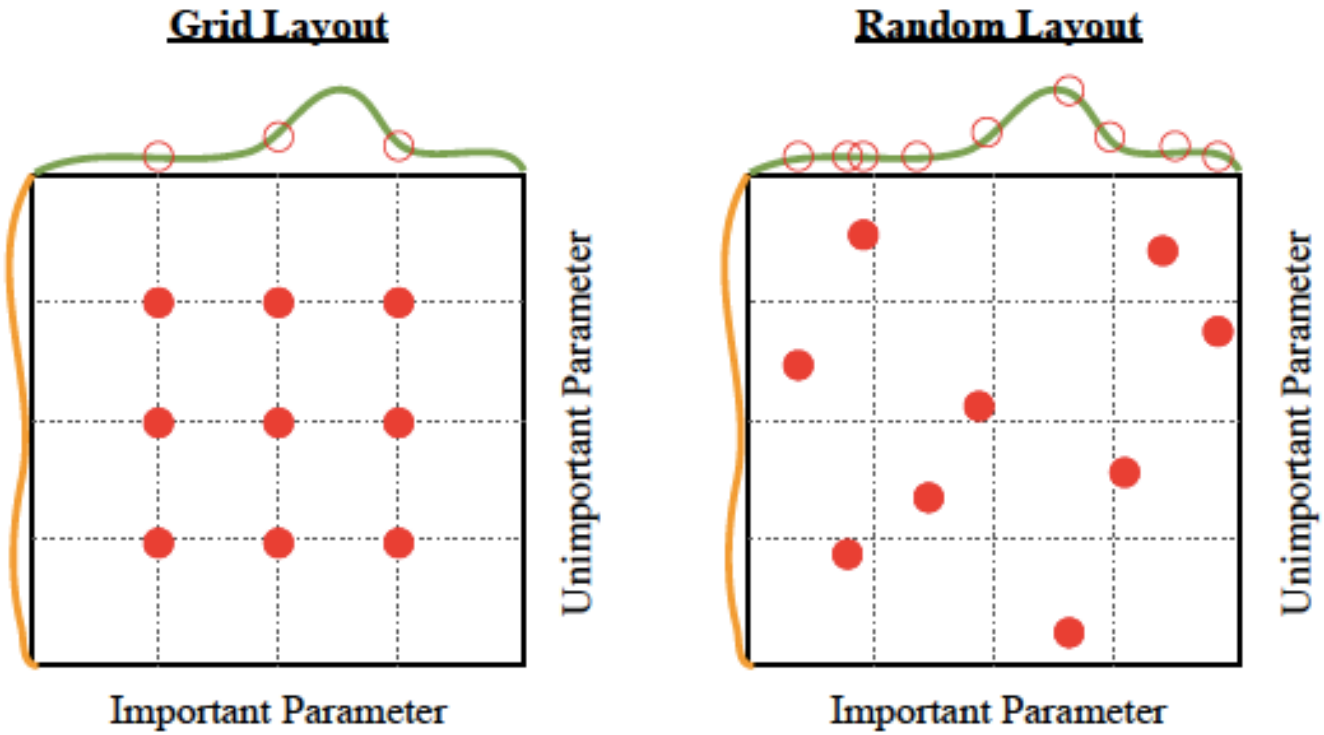
Learning rate, dropout probability, number of layers, neurons per layer…



Neural architecture search: too expensive

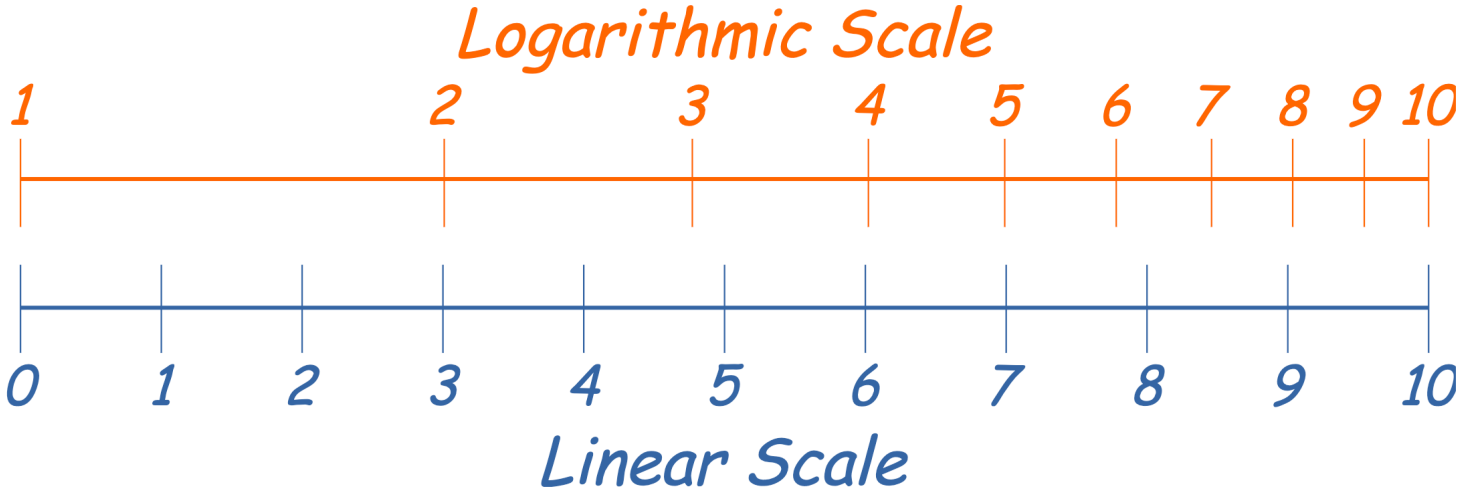
Hyperparameter tuning

Trick 1: Random search instead of grid search



Hyperparameter tuning

Trick 2: log-scale search



Training neural networks

Stochastic gradient descent

Dropout

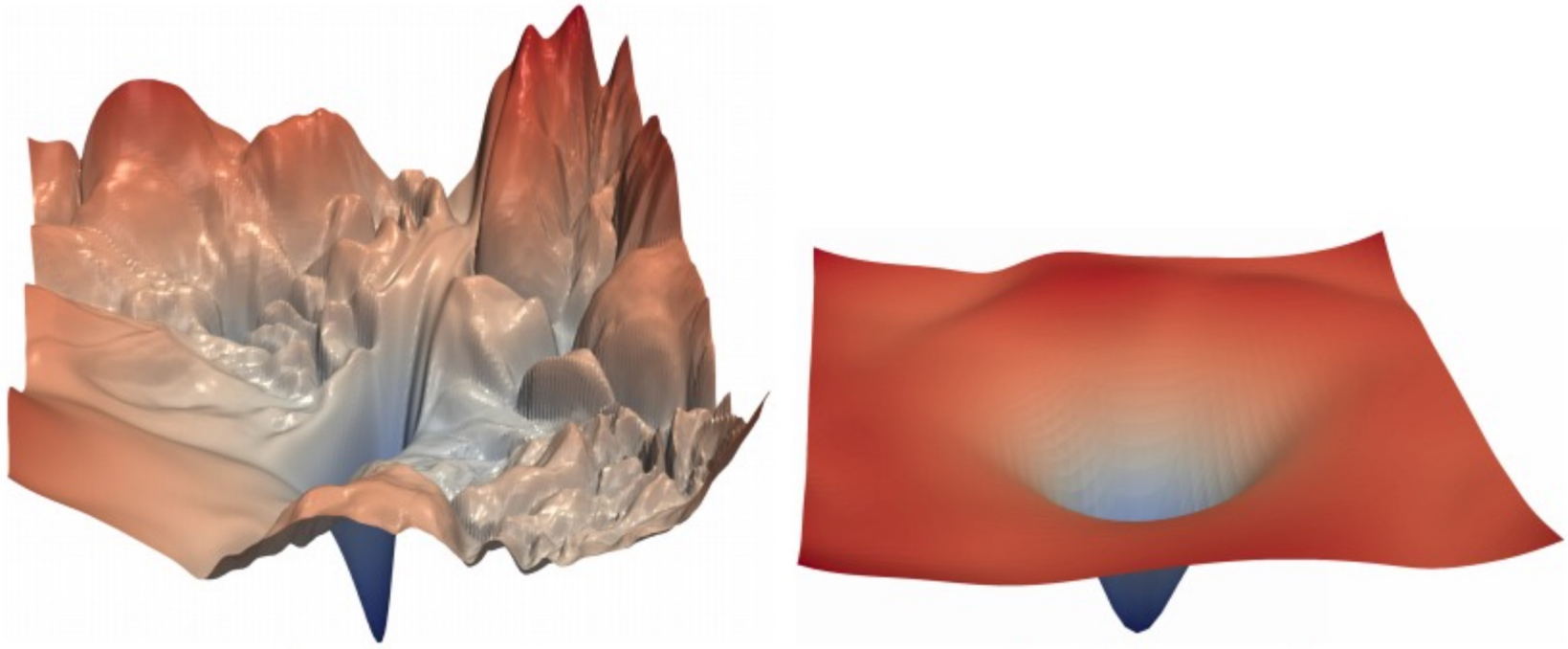
Initialization

Hyperparameter tuning

Batch normalization

Batch normalization

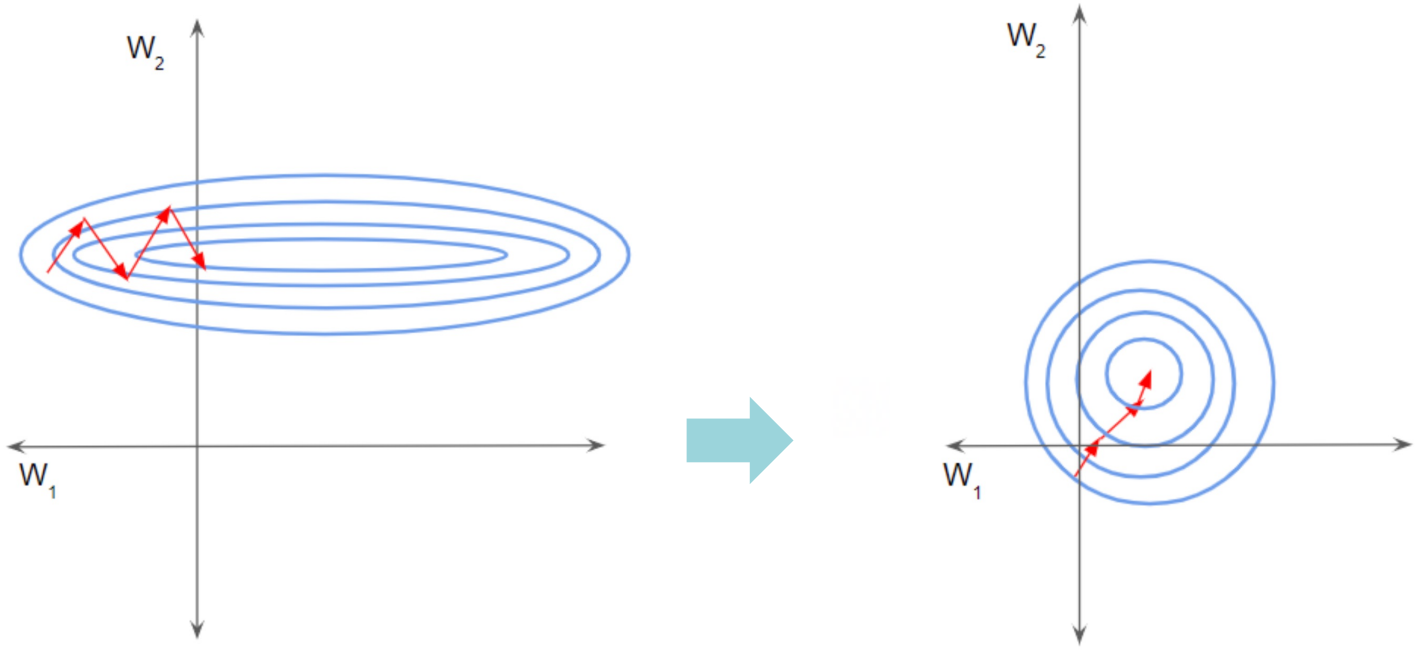
Make the loss landscape nicer



http://www.cs.cmu.edu/~mgormley/courses/10601/slides/lecture11-nn.pdf

Batch normalization

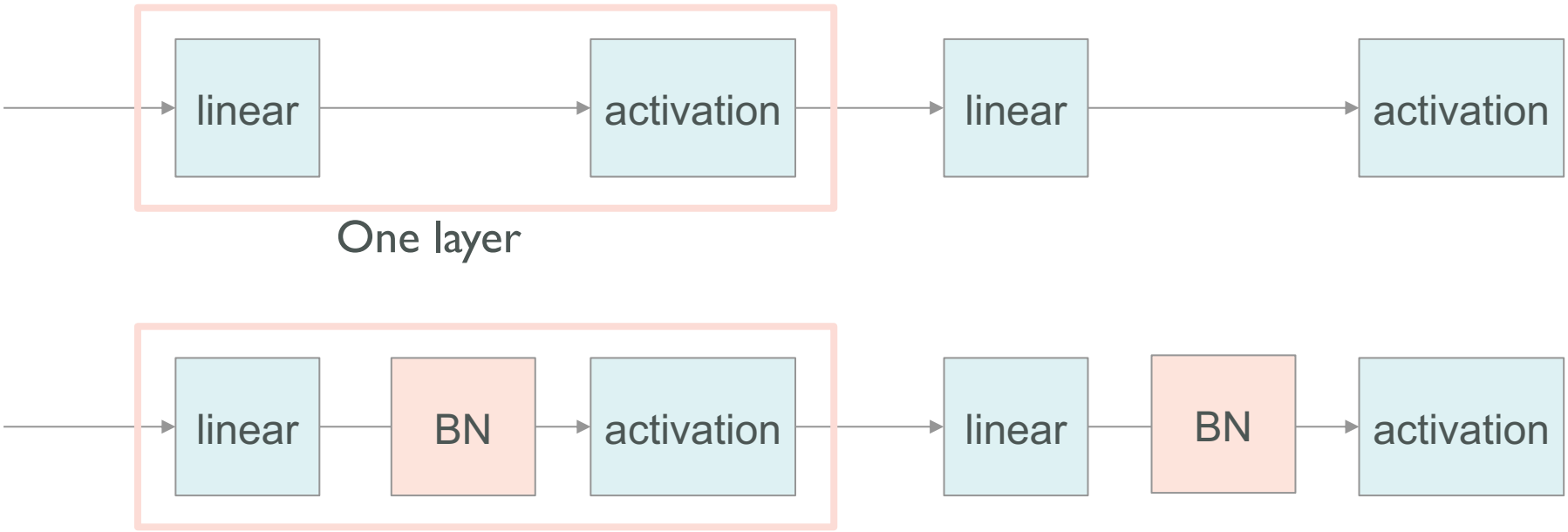
Make the loss landscape nicer



https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Batch normalization

Batch normalization is a method used to make neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.



Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

Training neural networks

Stochastic gradient descent: Use Adam

Dropout: Use when overfitting

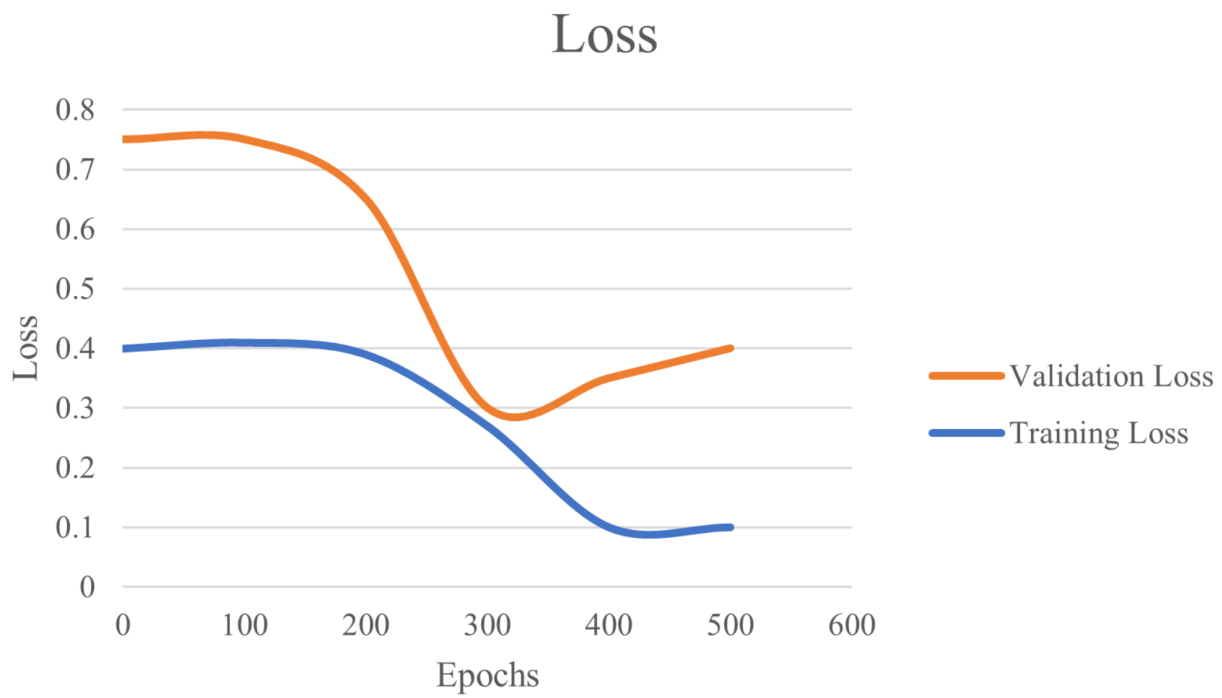
Initialization: Use Xavier initialization

Hyperparameter tuning: Random search

Batch normalization: Use all the time

Training neural networks

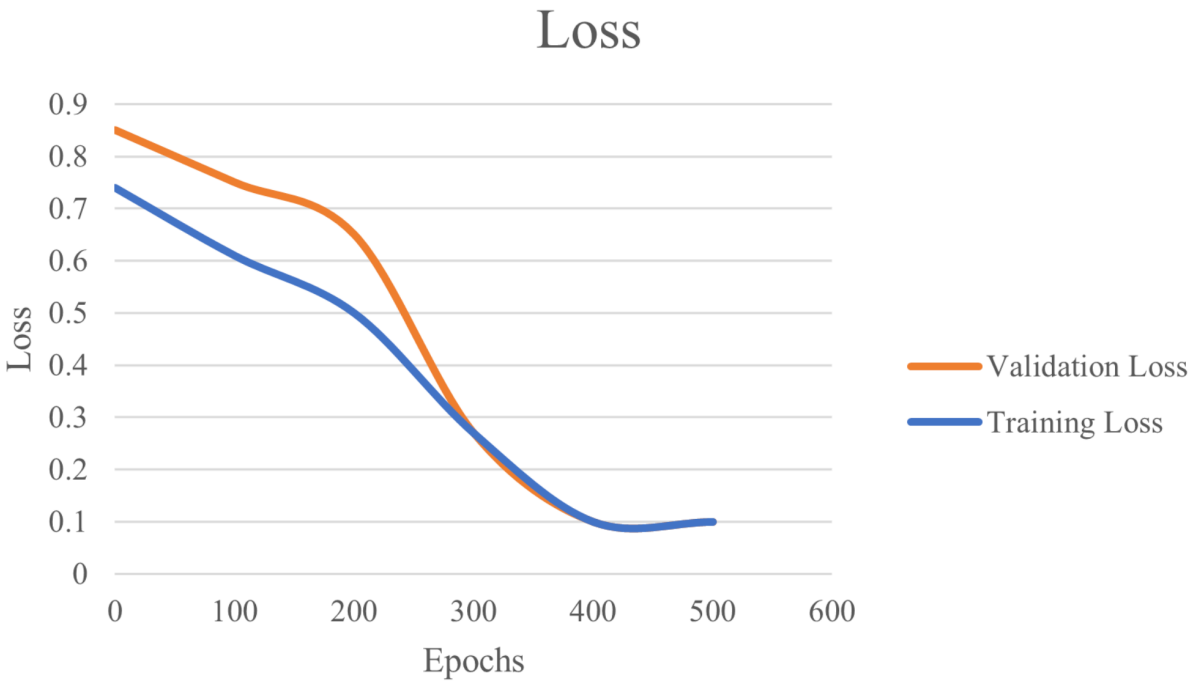
1. Train on a small dataset (overfitting: loss and visualization)



Training neural networks

1. Train on a small dataset (overfitting: loss and visualization)

2. Train on a large dataset (monitor your training/validation loss)



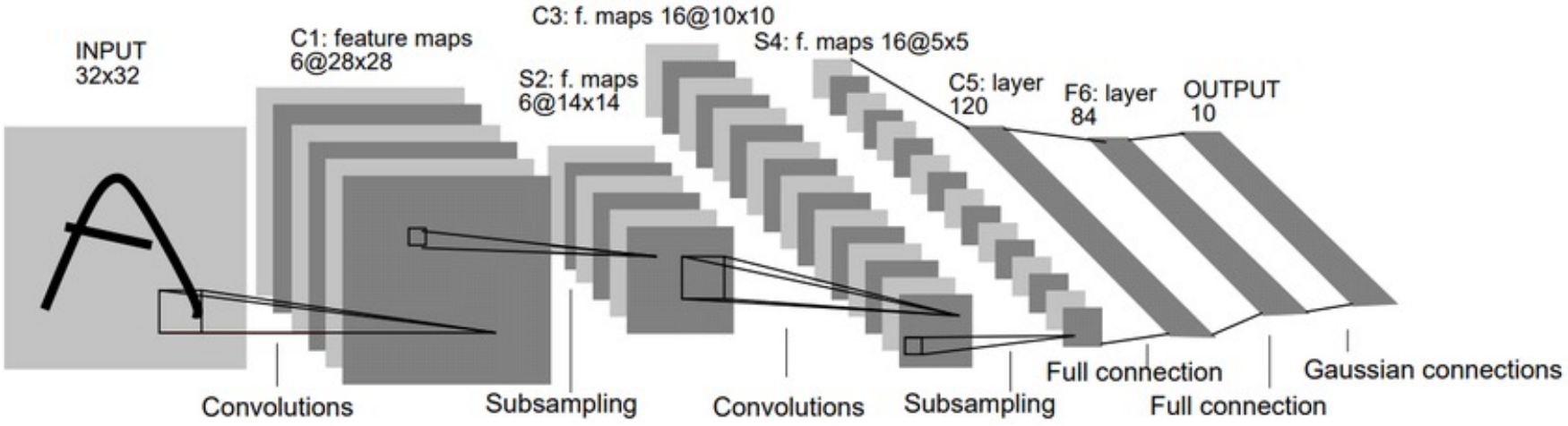
Training neural networks

1. Train on a small dataset (overfitting: loss and visualization)

2. Train on a large dataset (monitor your training/validation loss)

3. Evaluation metric and test

Next lecture: CNNs



Thank you very much!

sihengc@sjtu.edu.cn