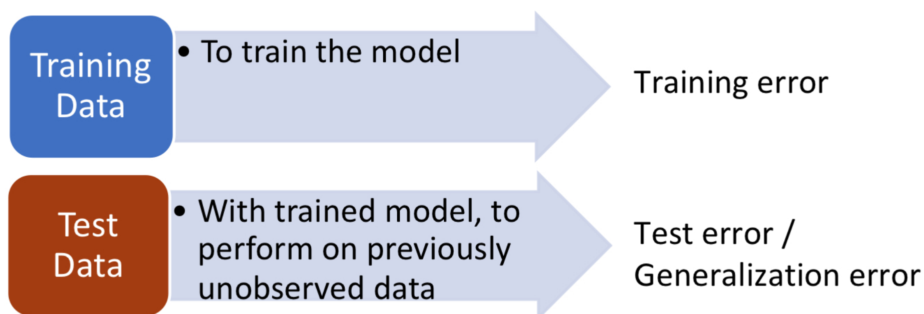
Deep Learning and Its Applications in Signal Processing

Lesson 2: Regularization and Optimization for Deep Learning

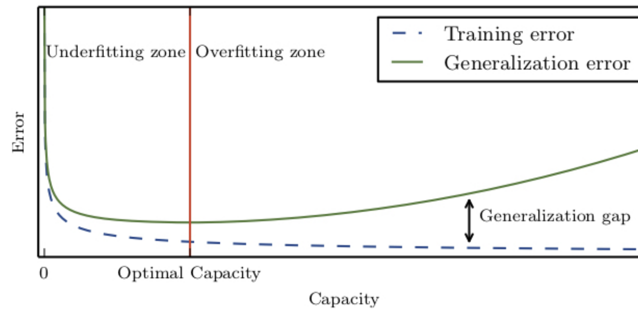
Liang Dong, ECE

Generalization



- Generalization: The model is capable of performing on previously unobserved input data.

Generalization: Underfitting and Overfitting

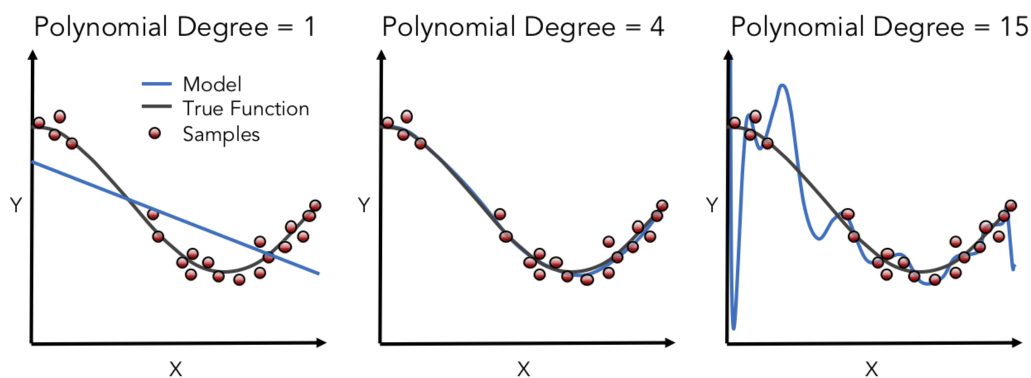


- Training error: Model error measured on the training data.
- Generalization error: Model error when it performs on new input data.

- Underfitting: both errors are high.
- Overfitting: generalization error increases and generalization gap widens.

Image from Deep Learning, by Goodfellow, Bengio, and Courville, The MIT Press, 2016.

Underfitting and Overfitting

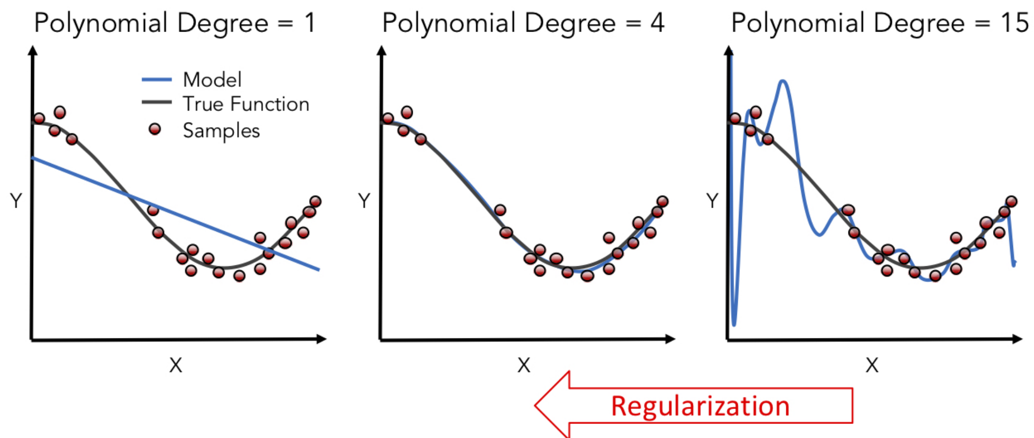


Underfitting
- Large training error and large gen. error
- High bias, low var

Matched
- Small gen. error
- "Just right"

Overfitting
- Very small training error but large gen. error
- Low bias, high var

Regularization



- Regularization: Take a model from "Overfitting" to "Matched".
- An effective regularizer reduces the variance significantly while not overly increasing the bias.

Parameter Norm Penalties

Add a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the cost function J to limit the capacity of the model.

$$\tilde{J}(\mathbf{X}, \mathbf{y}; \boldsymbol{\theta}) = J(\mathbf{X}, \mathbf{y}; \boldsymbol{\theta}) + \lambda \Omega(\boldsymbol{\theta})$$

where λ is a hyperparameter that weights the relative contribution of the norm penalty to the standard cost function.

- Typically, only the weights of the affine transformation are penalized but not the biases. Regularizing the biases can introduce a significant amount of underfitting

- Sometimes, we can use a different λ for each layer of the network.

Parameter Norm Penalties

- L^2 parameter regularization – weight decay

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|_2^2 = \frac{1}{2}\mathbf{w}^H\mathbf{w}$$

$$\nabla_{\mathbf{w}}\Omega(\mathbf{w}) = \mathbf{w}$$

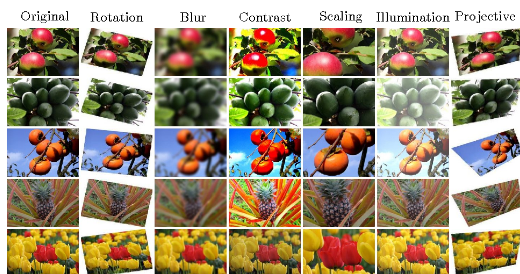
- L^1 parameter regularization

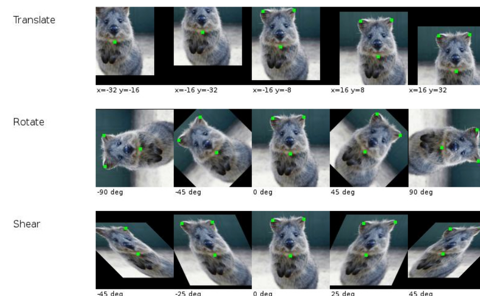
$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$$

$$\nabla_{\mathbf{w}}\Omega(\mathbf{w}) = \mathrm{sign}(\mathbf{w})$$

L^1 regularization results in a solution that is more sparse, i.e., some parameters have an optimal value of zero.
It can be used for feature selection.

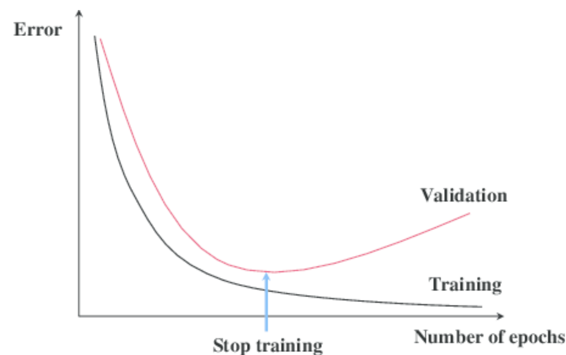
Dataset Augmentation as Regularization





- Train the model on more data by creating fake data and adding it to the training set.
- Useful for classification – The main task of a classifier is invariant to a wide variety of transformations.
- Inject noise in the input data, the hidden units, the weights, or the output targets.

Early Stopping



- Training error decreases steadily over time but validation error begins to rise.
- We obtain a better model by stopping at (returning to) the parameter setting at the point in time with the lowest validation error.

Parameter Sharing

- Model A and Model B deal with similar tasks and the model parameters may be close to each other. Therefore, we can use a parameter norm penalty

$$\Omega(\mathbf{w}^{(A)}, \mathbf{w}^{(B)}) = \|\mathbf{w}^{(A)} - \mathbf{w}^{(B)}\|_2^2$$

- Parameter sharing – Sets of parameters to be equal, e.g., in a convolutional neural network (CNN).

Sparse Representation

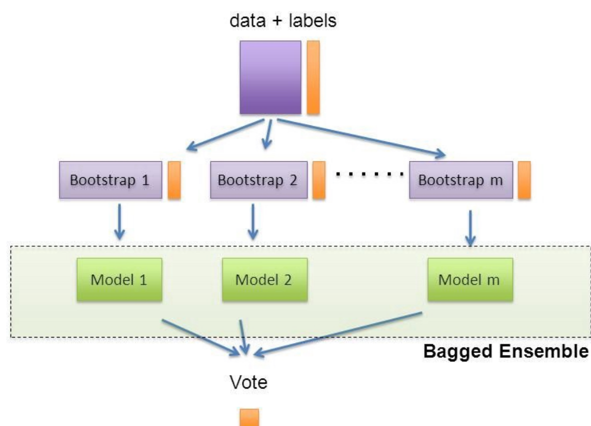
$$\underbrace{\begin{bmatrix} 11 \\ 18 \\ 6 \\ 1 \end{bmatrix}^{d\times 1}}_{\mathbf{y}} = \begin{bmatrix} 2 & 4 & -1 & 4 & -3 & 6 \\ 3 & 9 & -2 & 5 & 0 & 4 \\ 6 & 2 & 1 & 1 & -2 & -3 \\ 5 & 1 & 2 & 4 & 1 & -4 \end{bmatrix}^{d\times n} \underbrace{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}^{n\times 1}}_{\mathbf{h}}$$

- $\mathbf{h}$ is a sparse representation of data $\mathbf{x}$. Usually, $\mathbf{h}$ has k non-zero elements with $k \ll d$.
- The regularized cost function is

$$\tilde{J}(\mathbf{X}, \mathbf{y}; \boldsymbol{\theta}) = J(\mathbf{X}, \mathbf{y}; \boldsymbol{\theta}) + \lambda\Omega(\mathbf{h})$$

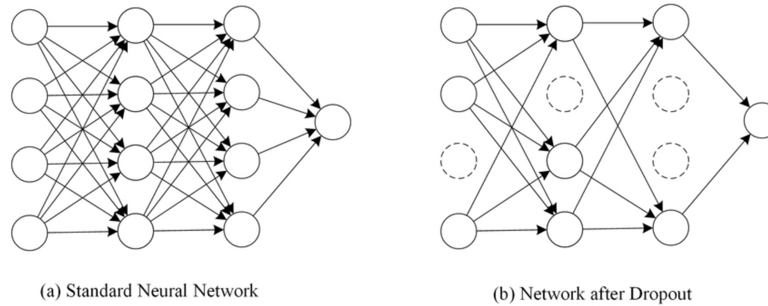
  The norm penalty on the representation is $\Omega(\mathbf{h}) = \|\mathbf{h}\|_1 = \sum_i |h_i|$.

Bagging – Bootstrap Aggregating



- Train several different models separately with different datasets from the original dataset.
- Usually, different models do not make the same error on the test data.
- All the models vote on the output for the test data.
  Model averaging / ensemble method

Dropout



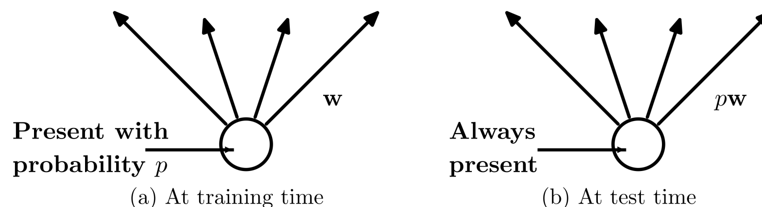
(a) Standard Neural Network

(b) Network after Dropout

- Dropout trains the ensemble consisting of subnetworks that can be formed by removing nonoutput units from the base network.
- At each training iteration (batch), we randomly remove a subset of input/hidden neurons with mask $\boldsymbol{\mu}$.
- Training:

$$\min \mathbb{E}_{\boldsymbol{\mu}} J(\boldsymbol{\theta}, \boldsymbol{\mu})$$

Dropout



(a) At training time

(b) At test time

- At test time we "rescale" the weight of the neuron to reflect the percentage of the time it was active.
- Inference:

$$p(y \mid \mathbf{x}) = \sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \mathbf{x}, \boldsymbol{\mu})$$

  where $p(\boldsymbol{\mu})$ is the probability distribution of sampling $\boldsymbol{\mu}$ at training time.
- Dropout trains a bagged ensemble of models that share hidden units.

Optimization for Learning

- Minimize the expected loss on the training set (the empirical risk)

$$\mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}(\mathbf{x},y)}L(f(\mathbf{x};\boldsymbol{\theta}),y) = \frac{1}{m}\sum_{i=1}^{m}L(f(\mathbf{x}^{(i)};\boldsymbol{\theta}),y^{(i)})$$

  where $\hat{p}_{\text{data}}(\mathbf{x},y)$ is the empirical distribution defined by the training set and m is the number of training samples.

- Minimize a surrogate loss function, e.g., the negative log-likelihood of the correct class. (Early stopping to prevent overfitting.)

- Stochastic/mini-batch gradient descent algorithms on a stream of data (online learning) help minimize the generalization error

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim p_{\text{data}}(\mathbf{x},y)}L(f(\mathbf{x};\boldsymbol{\theta}),y)$$

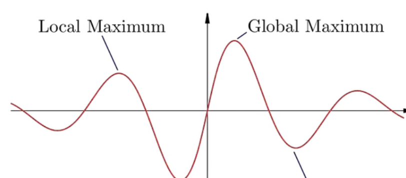
Challenges in Neural Network Optimization

- Neural networks are nonconvex functions that may have local minima.

- Model nonidentifiability – Any large training set cannot rule out all but one setting of the model's parameters. $\rightarrow$ Many local minima

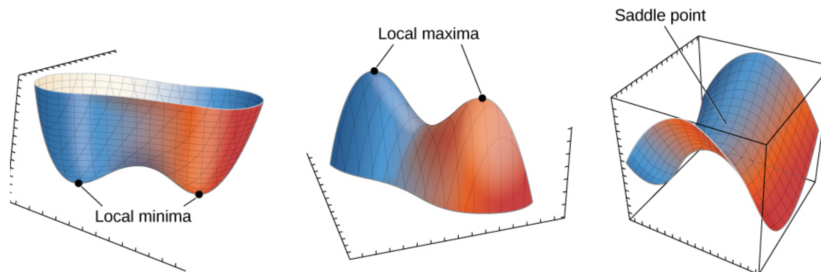
- Nevertheless, for sufficiently large neural networks, most local minima have a low cost function value compared with the global minimum. As a result, these local minima are not problematic.

- Find a point in the parameter space that has low but not necessarily minimal cost.



Challenges in Neural Network Optimization

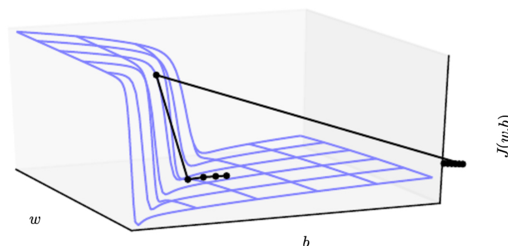
- Saddle points: In low-dimensional spaces, local minima are common. In higher-dimensional spaces, saddle points become more common.
- Gradient descent seems able to escape saddle points in many cases.



- Maxima are much like saddle points from the perspective of optimization.
- Flat regions: Gradient and Hessian are zero.

Challenges in Neural Network Optimization

- Cliff: Neural networks with many layers may have extremely steep regions. These result from the multiplication of several large weights or repeated multiplication by the same weight matrix $\mathbf{W}$, e.g., in a recurrent neural network (RNN).



- The gradient update step may move the parameters extremely far, i.e., jumping off the cliff. $\rightarrow$ Make learning unstable
- Vanishing and exploding gradient problem

Challenges in Neural Network Optimization

- Ill-conditioned Hessian matrix $\rightarrow$ Stochastic gradient descent gets stuck.
- Noisy and biased estimate of gradient and Hessian matrix.
- Intractable cost function as well as intractable gradient.
- Cost function lacks a global minimum point.

- Gradient descent is effective for making small local moves. Therefore, it is critical to choose good initial points.

Stochastic Gradient Descent (SGD)

- Compute gradient estimate $\hat{\mathbf{g}}$ from a mini-batch of m samples.

$$\hat{\mathbf{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$$

- Update parameters $\boldsymbol{\theta}$ with learning rate α_k at iteration k.

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha_k \hat{\mathbf{g}}$$

  The learning rate is gradually decreased to combat noise of SGD random sampling. e.g., with $\alpha_0 = 100\alpha_\tau$,

$$\alpha_k = \begin{cases} (1-\beta)\alpha_0 + \beta\alpha_\tau, \;\; \beta = k/\tau & 0 \leq k < \tau \\ \alpha_\tau & k \geq \tau \end{cases}$$

- Data shuffling after a training epoch. Changing the mini-batch size.

Momentum

- The method of momentum is designed to accelerate learning.
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

$$\begin{aligned}\mathbf{v} &= \beta\mathbf{v} - \alpha\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\boldsymbol{\theta}),\mathbf{y}^{(i)})\right] \\ \boldsymbol{\theta} &= \boldsymbol{\theta} + \mathbf{v}\end{aligned}$$

  where $\mathbf{v}$ is the velocity (or momentum with unit mass). $\beta \in [0, 1]$ determines how quickly the contributions of previous gradients decay.
- The gradient is a force that pushes the particle downhill along the cost function surface. β corresponds to a viscous drag on the movement.

Nesterov Momentum

- Nesterov's accelerated gradient method

$$\begin{aligned}\mathbf{v} &= \beta\mathbf{v} - \alpha\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\boldsymbol{\theta} + \beta\mathbf{v}),\mathbf{y}^{(i)})\right] \\ \boldsymbol{\theta} &= \boldsymbol{\theta} + \mathbf{v}\end{aligned}$$

- The gradient is evaluated after the current velocity is applied, i.e., adding a correction factor to the standard method of momentum.

Parameter Initialization

- The choice of initial point affects the training of deep models.
- Use random initialization to break symmetry between different units. i.e., If two hidden units with the same activation function are connected to the same inputs, they have different initial parameters.
- Randomly initialize the weights (with Gaussian or uniform distribution) but set the biases with heuristically chosen constants.
- Sometimes, we can initialize the weight matrix with a random orthogonal matrix.

Adaptive Learning Rates – AdaGrad

- Learning rate is a hyper-parameter that significantly affects model performance.
- Scale the learning rates inversely proportional to the square root of the sum of all the historical squared values of the gradient.

$$\begin{aligned} \mathbf{r} &= \mathbf{r} + \mathbf{g}^2 \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \frac{\alpha}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g} \end{aligned}$$

  where $\mathbf{g}$ is the gradient, α is a global learning rate, δ is a small number for numerical stability, $\cdot^2$, $\sqrt{\cdot}$ and $\odot$ are element-wise square, square root, and multiplication.
- Parameter with large gradient corresponds to a rapid decrease in the learning rate.

Adaptive Learning Rates – RMSProp

- RMSProp modifies AdaGrad to perform better for nonconvex cost function.
- Gradient accumulation with an exponentially weighted moving average. $\rightarrow$ To discard remote history.

$$
\begin{aligned}
\mathbf{r} &= \rho \mathbf{r} + (1-\rho)\mathbf{g}^2 \\
\boldsymbol{\theta} &= \boldsymbol{\theta} - \frac{\alpha}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}
\end{aligned}
$$

- RMSProp combined with Nesterov momentum

$$
\begin{aligned}
\mathbf{r} &= \rho \mathbf{r} + (1-\rho)\mathbf{g}^2 \\
\mathbf{v} &= \beta \mathbf{v} - \frac{\alpha}{\sqrt{\mathbf{r}}} \odot \mathbf{g} \\
\boldsymbol{\theta} &= \boldsymbol{\theta} + \mathbf{v}
\end{aligned}
$$

Adaptive Learning Rates – Adam (Adaptive Moments)

- Estimates of the first-order and second-order moments.
- Bias corrections to the estimates of the moments.

Initialize first-order and second-order moments $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$. $t = 0$.

$$
\begin{aligned}
t &= t + 1 \\
\mathbf{s} &= \rho_1 \mathbf{s} + (1-\rho_1)\mathbf{g} \\
\mathbf{r} &= \rho_2 \mathbf{r} + (1-\rho_2)\mathbf{g}^2 \\
\hat{\mathbf{s}} &= \frac{\mathbf{s}}{1-\rho_1^t} \\
\hat{\mathbf{r}} &= \frac{\mathbf{r}}{1-\rho_2^t} \\
\boldsymbol{\theta} &= \boldsymbol{\theta} - \frac{\alpha \hat{\mathbf{s}}}{\delta + \sqrt{\hat{\mathbf{r}}}} \odot \mathbf{g}
\end{aligned}
$$

Repeat.

Newton's Method

- Newton's method for optimization uses the second-order Hessian $\mathbf{H}$.

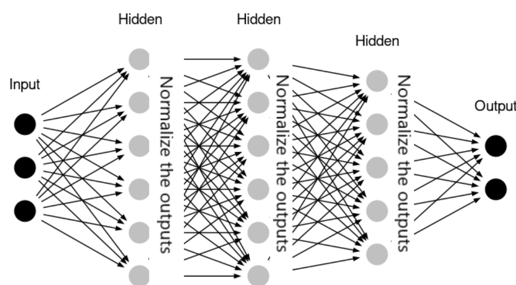
$$
\begin{aligned}
\mathbf{g} &= \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right] \\
\mathbf{H} &= \nabla^2_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right] \\
\boldsymbol{\theta} &= \boldsymbol{\theta} - \mathbf{H}^{-1}\mathbf{g}
\end{aligned}
$$

- If the Hessian is not positive definite, use regularized update

$$
\boldsymbol{\theta} = \boldsymbol{\theta} - (\mathbf{H} + \alpha \mathbf{I})^{-1}\mathbf{g}
$$

- Computationally complex with the inversion of Hessian $\mathbf{H}$.

Batch Normalization



- Batch normalization can be applied to any input or hidden layer to resolve the problem that the parameter update for one layer affects other layers.

- Batch normalization allows each layer of a network to learn by itself a little more independently of other layers.

- Batch normalization reduces overfitting because it adds some noise to each hidden layer's activation outputs.

Batch Normalization

- Mini-batch mean: $\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathbf{x}_i$
  where m is the size of the mini batch.

- Mini-batch variance: $\boldsymbol{\sigma}^2 = \frac{1}{m} \sum_i (\mathbf{x}_i - \boldsymbol{\mu})^2$ (element-wise arithmetic)

- Batch Normalization: $\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \boldsymbol{\mu}}{\sqrt{\delta + \boldsymbol{\sigma}^2}}$ (element-wise arithmetic)

- SGD does the "denormalization": $\mathbf{y}_i = \gamma \hat{\mathbf{x}}_i + \beta$

- Batch normalization adds two trainable parameters γ and β to each layer.