Unit-3

Deep Neural Networks

Contents

- Backpropagation
- Setup and Initialization Issues
- Gradient-Descent strategies
- Bias-variance trade-off
- Generalization Issues in Model Tuning and Evaluation
- Ensemble Methods

Backpropagation

- Back propagation is a specific technique for implementing gradient descent in weight space for a multilayer perceptron.
- The basic idea is to efficiently compute partial derivatives of an approximating function F(w, x) realized by the network w.r.t
  - all the elements of the adjustable weight vector w for a given value of input vector x

Backpropagation: Two passes of computation

- In the application of the back-propagation algorithm, two different passes of computation are distinguished.
- The first pass is referred to as the forward pass, and the second is referred to as the backward pass.

1. In the forward pass, the synaptic weights remain unaltered throughout the network, and the function signals (activation values) of the network are computed on a neuron-by-neuron basis.

Backpropagation: Two passes of computation

2. The backward pass starts at the output layer by passing the error signals leftward through the network, layer by layer, and recursively computing the the local gradient for each neuron.
   - This local gradient is useful to update the weights.

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*, or

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)}$$

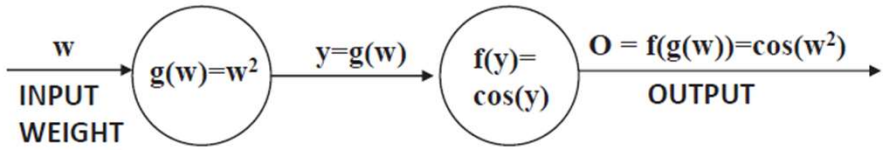
$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning-} \\ \textit{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j, \\ y_i(n) \end{pmatrix}$$

Backpropagation: Local gradient computation

- The local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi`_j(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j;
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi`_j(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron j;

Backpropagation along Single Path (Univariate Chain Rule)



- Consider a two-node path with $f(g(w)) = \cos(w^2)$
- In the univariate chain rule, we compute product of *local* derivatives.

$$\frac{\partial f(g(w))}{\partial w} = \underbrace{\frac{\partial f(y)}{\partial y}}_{-sin(y)} \cdot \underbrace{\frac{\partial g(w)}{\partial w}}_{2w} = -2w \cdot \sin(y) = -2w \cdot \sin(w^2)$$

- Local derivatives are easy to compute because they care about their own input and output.

Backpropagation along Multiple Paths (Multivariate Chain Rule)

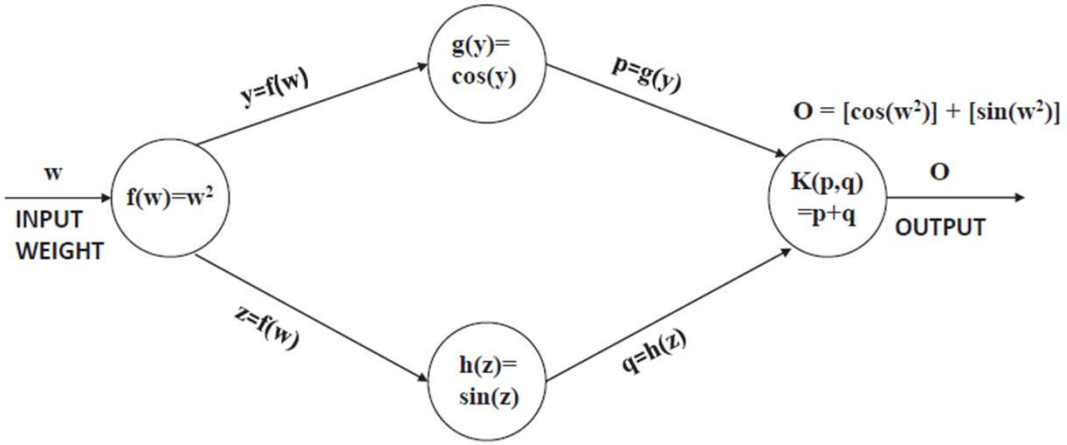
- Neural networks contain multiple nodes in each layer.

- Consider the function $f(g_1(w), \ldots g_k(w))$, in which a unit computing the *multivariate* function $f(\cdot)$ gets its inputs from k units computing $g_1(w) \ldots g_k(w)$.

- The *multivariable chain rule* needs to be used:

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \quad (2)$$

Example of Multivariable Chain Rule



$$\frac{\partial o}{\partial w} = \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w}$$

$$= -2w \cdot \sin(y) + 2w \cdot \cos(z)$$

$$= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$$

- Product of local derivatives along *all* paths from w to o.

Pathwise Aggregation Lemma

- Let a non-null set $\mathcal{P}$ of paths exist from a variable w in the computational graph to output o.
  - Local gradient of node with variable $y(j)$ with respect to variable $y(i)$ for directed edge (i, j) is $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$
- The value of $\frac{\partial o}{\partial w}$ is given by computing the product of the local gradients along each path in $\mathcal{P}$, and summing these products over all paths.

$$\frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{3}$$

- Observation: Each $z(i, j)$ easy to compute.

Dynamic Programming and Directed Acyclic Graphs

- Dynamic programming used extensively in directed acyclic graphs.
  - **Typical:** Exponentially aggregative path-centric functions between source-sink pairs.
  - **Example:** Polynomial solution to longest path problem in directed acyclic graphs (NP-hard in general).
  - **General approach:** Starts at either the source or sink and *recursively* computes the relevant function over paths of increasing length by reusing intermediate computations.
- Our path-centric function: $S(w, o) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j)$.
  - Backwards direction makes more sense here because we have to compute derivative of output (sink) with respect to all variables in early layers.

Dynamic Programming Update

- Let $A(i)$ be the set of nodes at the ends of outgoing edges from node i.

- Let $S(i, o)$ be the *intermediate* variable indicating the same path aggregative function from i to o.

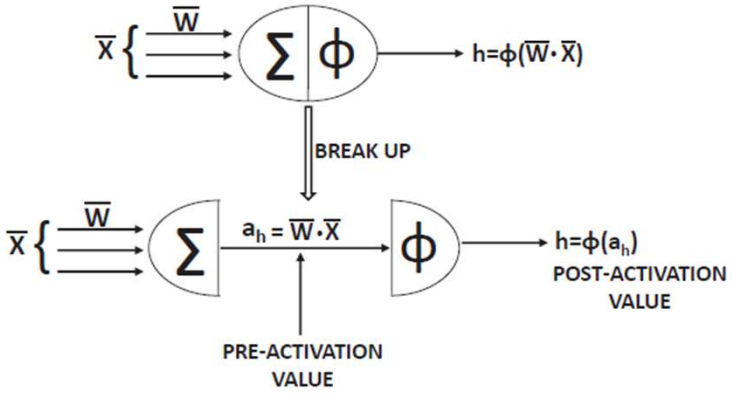
$$S(i, o) \Leftarrow \sum_{j \in A(i)} S(j, o) \cdot z(i, j) \tag{4}$$

- Initialize $S(o, o)$ to 1 and compute backwards to reach $S(w, o)$.
  - Intermediate computations like $S(i, o)$ are also useful for computing derivatives in other layers.

- Do you recognize the multivariate chain rule in Equation 4?

$$\frac{\partial o}{\partial y(i)} = \sum_{j \in A(i)} \frac{\partial o}{\partial y(j)} \cdot \frac{\partial y(j)}{\partial y(i)}$$

How Does it Apply to Neural Networks?



- A neural network is a special case of a computational graph.
  - We can define the computational graph in multiple ways.
  - Pre-activation variables or post-activation variables or both as the node variables of the computation graph?
  - The three lead to different updates but the end result is equivalent.

Pre-Activation Variables to Create Computational Graph

- Compute derivative $\delta(i, o)$ of loss L at o with respect to pre-activation variable at node i.

- We always compute loss derivatives $\delta(i, o)$ with respect to activations in *nodes* during dynamic programming rather than *weights*.

  - Loss derivative with respect to weight w_{ij} from node i to node j is given by the product of $\delta(j, o)$ and hidden variable at i (why?)

- Key points: $z(i, j) = w_{ij} \cdot \Phi_i'$, Initialize $S(o, o) = \delta(o, o) = \frac{\partial L}{\partial o} \Phi_o'$

$$\delta(i, o) = S(i, o) = \Phi_i' \sum_{j \in A(i)} w_{ij} S(j, o) = \Phi_i' \sum_{j \in A(i)} w_{ij} \delta(j, o) \tag{5}$$

Backpropagation with Pre-activation Variables

The backpropagation process can now be described as follows:

1. Use a forward-pass to compute the values of all hidden units, output o, and loss L for a particular input-output pattern (X, y).

2. Initialize $\partial L / \partial a_o = \delta(o, o)$ to $\partial L / \partial o \cdot \Phi`(a_o)$.

3. Compute each $\delta(h_r, o)$ in the backwards direction. After each such computation, compute the gradients with respect to incident weights as follows:

$$\partial L / \partial w(h_{r-1}, h_r) = \delta(h_r, o) \cdot h_{r-1}$$

- The partial derivatives with respect to incident biases can be computed by using the fact that bias neurons are always activated at a value of +1.

4. Use the computed partial derivatives of loss function with respect to weights in order to perform stochastic gradient descent for input-output pattern (X, y).

Post-Activation Variables to Create Computation Graph

- The variables in the computation graph are hidden values *after* activation function application.

- Compute derivative $\Delta(i, o)$ of loss L at o with respect to post-activation variable at node i.

- Key points: $z(i, j) = w_{ij} \cdot \Phi'_j$, Initialize $S(o, o) = \Delta(o, o) = \frac{\partial L}{\partial o}$

$$\Delta(i, o) = S(i, o) = \sum_{j \in A(i)} w_{ij} S(j, o) \Phi'_j = \sum_{j \in A(i)} w_{ij} \Delta(j, o) \Phi'_j \tag{6}$$

  - Compare with pre-activation approach $\delta(i, o) = \Phi'_i \sum_{j \in A(i)} w_{ij} \delta(j, o)$

Local Gradient Updates for various Activation Functions

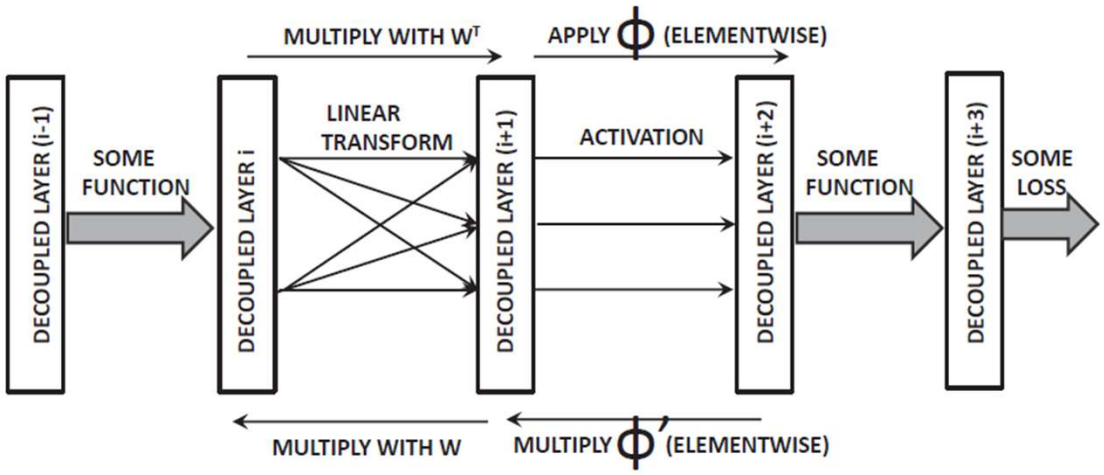
$$\delta(h_r, o) = \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \delta(h, o) \quad \text{[Linear]}$$

$$\delta(h_r, o) = h_r(1 - h_r) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \delta(h, o) \quad \text{[Sigmoid]}$$

$$\delta(h_r, o) = (1 - h_r^2) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \delta(h, o) \quad \text{[Tanh]}$$

$$\delta(h_r, o) = \begin{cases} \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \delta(h, o) & \text{if } 0 < a_{h_r} \\ 0 & \text{otherwise} \end{cases} \quad \text{[ReLU]}$$

Effect on Linear Layer and Activation Functions



- Backpropagation is multiplication with transposed weight matrix for linear layer.

- Elementwise multiplication with derivative for activation layer.

Losses at Arbitrary Nodes

- We assume that the loss is incurred at a single output node.
- In case of multiple output nodes, one only has to add up the contributions of different outputs in the backwards phase.
- In some cases, penalties may be applied to hidden nodes.
- For a hidden node i, we add an "initialization value" to $S(i, o)$ just after it has been computed during dynamic programming, which is based on its penalty.
  - Similar treatment as the initialization of an output node, except that we *add* the contribution to existing value of $S(i, o)$.

Handling Shared Weights

1. In an autoencoder simulating PCA , the weights in the input layer and the output layer are shared.
2. In a recurrent neural network for text, the weights in different temporal layers are shared, because it is assumed that the language model at each time-stamp is the same.
3. In a convolutional neural network, the same grid of weights (corresponding to a visual field) is used over the entire spatial extent of the neurons

Mini-batch Stochastic Gradient Descent

- One can improve accuracy of gradient computation by using a batch of instances.
  - Instead of holding a vector of activations, we hold a matrix of activations in each layer.
  - Matrix-to-matrix multiplications required for forward and backward propagation.
  - Increases the memory requirements.
- Typical sizes are powers of 2 like 32, 64, 128, 256

Why Does Mini-Batching Work?

- At early learning stages, the weight vectors are very poor.
  - Training data is highly redundant in terms of important patterns.
  - Small batch sizes gives the correct direction of gradient.
- At later learning stages, the gradient direction becomes less accurate.
  - But some amount of noise helps avoid overfitting anyway!

Setup and Initialization Issues

- Tuning Hyperparameters:
  - Neural networks have a large number of hyperparameters
  - The term “hyperparameter” is used to specifically refer to the parameters regulating the design of the model
    - like learning rate
    - number of layers
    - nodes per layer
    - and regularization parameter

Setup and Initialization Issues

- Tuning Hyperparameters:
  - Hyperparameter tuning **based on validation set:** a portion of the data is held out as validation data, and the performance of the model is tested on the validation set with various choices of hyperparameters.
  - **Grid-based hyperparameter exploration:** select set of values for each parameter in some reasonable range.
    - Test over all combination of values
    - With 10 parameters, choosing just 3 values for each parameter leads to 3^{10} values
    - https://www.section.io/engineering-education/grid-search/#grid-search

Setup and Initialization Issues

- Tuning Hyperparameters:
  - **Sampling logarithm of hyperparameters:** Search uniformly in reasonable values of log-values and then exponentiate.
    - **Example:** Uniformly sample log-learning rate between −3 and −1, and then raise it to the power of 10.
  - In many cases, multiple threads of the process with different hyperparameters can be run, and one can successively terminate or add new sampled runs.
  - In the end, only one winner is allowed to train to completion.
    - Sometimes a few winners may be allowed to train to completion, and their predictions will be averaged as an ensemble.

Setup and Initialization Issues

- Feature preprocessing:
  - *Additive Preprocessing and mean-centering:* It can be useful to mean-center the data in order to remove certain types of bias effects.
  - In such cases, a vector of column-wise means is subtracted from each data point.
- *Non-negative features:* A second type of pre-processing is used when it is desired for all feature values to be non-negative.
  - In such a case, the absolute value of the most negative entry of a feature is added to the corresponding feature value of each data point.

Setup and Initialization Issues

- Feature preprocessing:
  - *Feature Normalization:* A common type of normalization is to divide each feature value by its standard deviation.
  - When this type of feature scaling is combined with mean-centering, the data is said to have been standardized.
  - Each feature is presumed to have been drawn from a standard normal distribution with zero mean and unit variance.
  - Min-max normalization: useful when the data needs to be scaled in the range (0,1)

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j}$$

Setup and Initialization Issues

- Feature preprocessing:
  - *Whitening:* The axis-system is rotated to create a new set of de-correlated features, each of which is scaled to unit variance.
  - Typically, principal component analysis is used to achieve this goal.
    - Principal component analysis can be viewed as the application of singular value decomposition after mean-centering a data matrix (i.e., subtracting the mean from each column).
    - Let D be an $n \times d$ data matrix that has already been mean-centered.

Setup and Initialization Issues

- Feature preprocessing:
  - *Whitening* Steps used for each data point:

i. The mean of each column is subtracted from the corresponding feature;

ii. Each d-dimensional row vector representing a training data point (or test data point) is post-multiplied with P to create a k-dimensional row vector;

iii. Each feature of this k-dimensional representation is divided by the square-root of the corresponding eigenvalue.

Initialization Issues

- Initialization is particularly important in neural networks because of the stability issues associated with neural network training.
- One possible approach to initialize the weights is to generate random values from a Gaussian distribution with zero mean and a small standard deviation.
  - Problem with this initialization is that it is not sensitive to the number of inputs to a specific neuron.

Initialization Issues

Sensitivity to Number of Inputs

- More inputs increase output sensitivity to the average weight.
  - Additive effect of multiple inputs: variance linearly increases with number of inputs r.
  - Standard deviation scales with the square-root of number of inputs r.
- Each weight is initialized from Gaussian distribution with standard deviation $\sqrt{1/r}$ ($\sqrt{2/r}$ for ReLU).
- **More sophisticated:** Use standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

Gradient Descent Strategies

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$
$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \nabla b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w=w_t, b=b_t}$$

Gradient Descent Algorithm

Algorithm 1: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

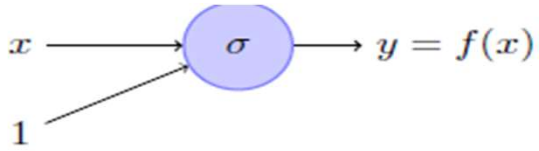
while $t < max\_iterations$ do

  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;

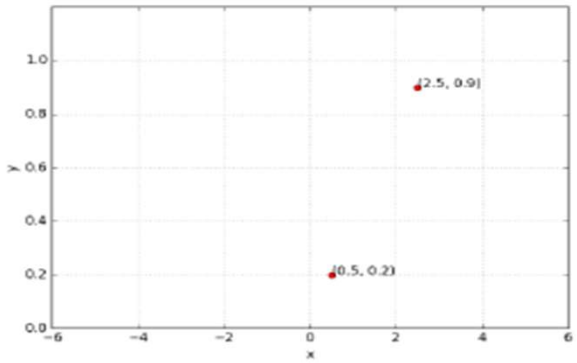
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;

end

Gradient Descent Algorithm



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

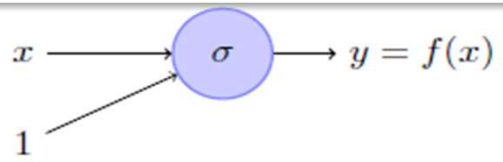


Let's assume there is only 1 point to fit (x, y)

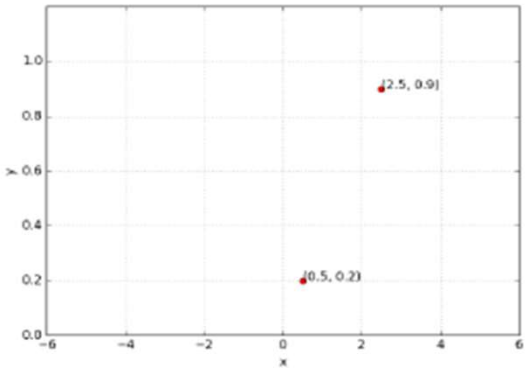
$$\mathscr{L}(w, b) = \frac{1}{2} * (f(x) - y)^2$$

$$\nabla w = \frac{\partial \mathscr{L}(w, b)}{\partial w} = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$

Gradient Descent Algorithm



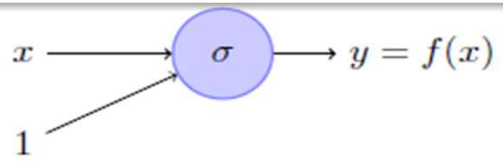
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



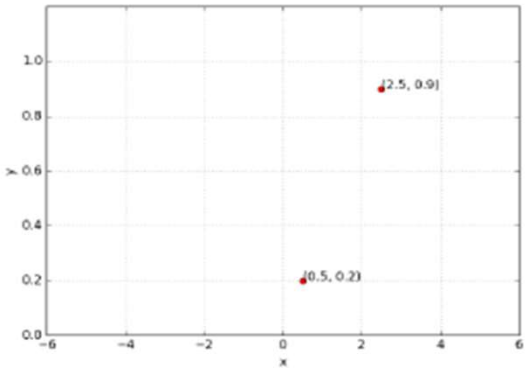
So if there is only 1 point (x, y), we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

Gradient Descent Algorithm



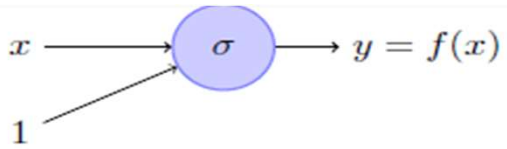
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



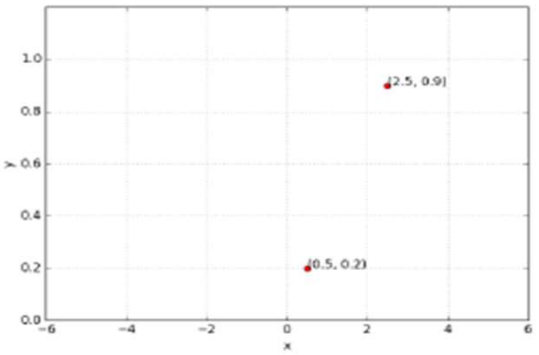
So if there is only 1 point (x, y), we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

Gradient Descent Algorithm



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



So if there is only 1 point (x, y), we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

Gradient Descent Algorithm with Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;
while $t{+}{+} < max\_iterations$ **do**
  $h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, \hat{y} = forward\_propagation(\theta_t)$;
  $\nabla\theta_t = backward\_propagation(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y})$;
  $\theta_{t+1} \leftarrow \theta_t - \eta\nabla\theta_t$;
end

Gradient Descent Algorithm with Backpropagation

Algorithm: forward_propagation(θ)

for $k = 1$ *to* $L - 1$ do
  $a_k = b_k + W_k h_{k-1}$;
  $h_k = g(a_k)$;
end
$a_L = b_L + W_L h_{L-1}$;
$\hat{y} = O(a_L)$;

Gradient Descent Algorithm with Backpropagation

Algorithm: back_propagation$(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y})$

//Compute output gradient ;
$\nabla_{a_L} \mathscr{L}(\theta) = -(e(y) - \hat{y})$;
for $k = L$ *to* 1 **do**
    // Compute gradients w.r.t. parameters ;
    $\nabla_{W_k} \mathscr{L}(\theta) = \nabla_{a_k} \mathscr{L}(\theta) h_{k-1}^T$;
    $\nabla_{b_k} \mathscr{L}(\theta) = \nabla_{a_k} \mathscr{L}(\theta)$;
    // Compute gradients w.r.t. layer below ;
    $\nabla_{h_{k-1}} \mathscr{L}(\theta) = W_k^T (\nabla_{a_k} \mathscr{L}(\theta))$;
    // Compute gradients w.r.t. layer below (pre-activation);
    $\nabla_{a_{k-1}} \mathscr{L}(\theta) = \nabla_{h_{k-1}} \mathscr{L}(\theta) \odot [\ldots, g'(a_{k-1,j}), \ldots]$;
end

Gradient Descent Strategies

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$

Gradient Descent Strategies

- A lower learning rate used early on will cause the algorithm to take too long to come even close to an optimal solution.
- A large initial learning rate will allow the algorithm to come reasonably close to a good solution at first;
- In either case, maintaining a constant learning rate is not ideal.
- **Allowing the learning rate to decay over time can naturally achieve the desired learning-rate adjustment to avoid these challenges.**

Learning Rate Decay

- Initial learning rates should be high but reduce over time.

- The two most common decay functions are *exponential decay* and *inverse decay*.

- The learning rate α_t can be expressed in terms of the initial decay rate α_0 and epoch t as follows:

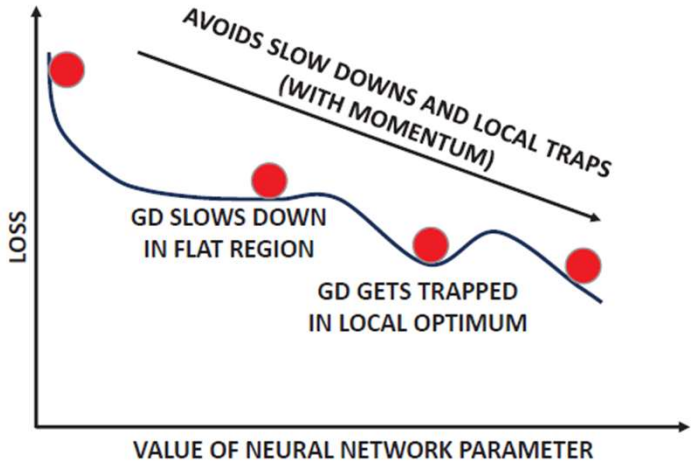
$$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$
$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

  The parameter k controls the rate of the decay.

First-order GD optimizer methods

- Momentum
- Nesterov Accelerated Gradient Momentum
- AdaGrad
- RMSProp
- AdaDelta
- Adam

Momentum Methods: Marble Rolling Down Hill



- Use a *friction parameter* $\beta \in (0, 1)$ to gain speed in direction of movement.

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Considering an exponentially weighted average

Momentum-based Gradient Descent

Intuition

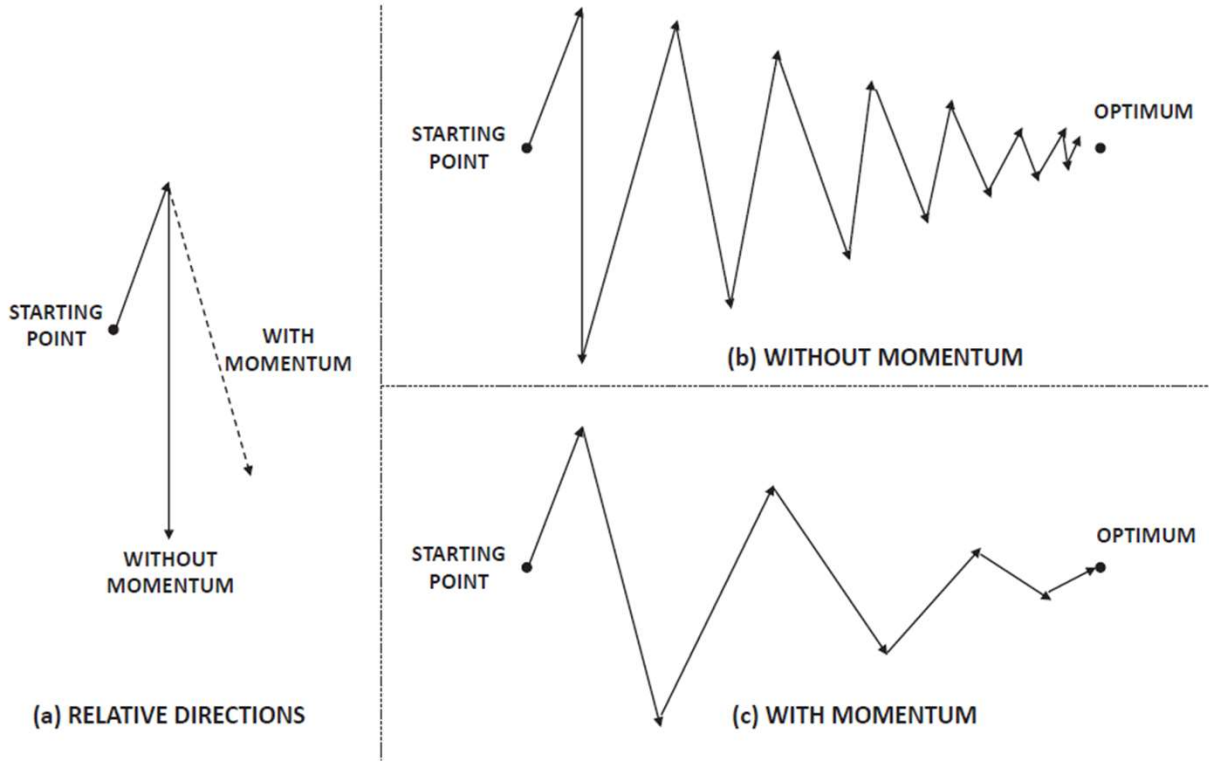
- If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction
- Just as a ball gains momentum while rolling down a slope

Update Rule:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Considering an exponentially weighted average

Avoiding Zig-Zagging with Momentum



Momentum

- Due to the momentum, the optimizer may overshoot a bit, then come back , overshoot again and oscillate like this many times before stabilizing at the minimum.
- This is why it is good to have bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.
  - So the hyperparameter, friction momentum β must be set between 0 (high friction) and 1(low friction)
  - Typically, β=0.9

Nesterov Momentum

- Modification of the traditional momentum method in which *the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again.*

- Compute at a point reached using only the momentum portion of the current update:

$$\overline{V} \Leftarrow \underbrace{\beta\overline{V}}_{\text{Momentum}} - \alpha\frac{\partial L(\overline{W}+\beta\overline{V})}{\partial\overline{W}}; \quad \overline{W} \Leftarrow \overline{W}+\overline{V}$$

[**Look ahead before Leap**]

- Put on the brakes as the marble reaches near bottom of hill.

- Nesterov momentum should always be used with mini-batch SGD (rather than SGD).

Nesterov Momentum/Nesterov Accelerated Gradient (NAG)

Observations about NAG

- Looking ahead helps NAG in correcting its course quicker than momentum based gradient descent
- Hence the oscillations are smaller and the chances of escaping the minima valley also smaller

Ill-conditioning

- **Ill-conditioning** is a situation where the loss function has an inherent tendency to be more sensitive to some parameters than others
  - for instance, after feature normalization
- **Extreme ill-conditioning:**
  - for which the partial derivatives of the loss are wildly different with respect to the different parameters.
  - Clever learning strategies exist that work well in these ill-conditioned settings.

AdaGrad

- *Aggregate* squared magnitude of ith partial derivative in A_i.
- The square-root of A_i is proportional to the root-mean-square slope.
  - The absolute value will increase over time.

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{15}$$

- The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i}\right); \quad \forall i \tag{16}$$

- Use $\sqrt{A_i + \epsilon}$ in the denominator to avoid ill-conditioning.

AdaGrad

- Intuition:
  - **Decay the learning rate for parameters in proportion to their update history (more updates means more decay)**
  - Scaling the derivative inversely with $\sqrt{A_i}$ encourages faster relative movements along gently sloping directions.
    – Absolute movements tend to slow down prematurely.
    =>AdaGrad may not converge
  - Cons: Adagrad decays the learning rate very aggressively (as the denominator grows)

RMSProp

- The RMSProp algorithm uses *exponential smoothing* with parameter $\rho \in (0, 1)$ in the relative estimations of the gradients.
  - Absolute magnitudes of scaling factors do not grow with time.

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L}{\partial w_i} \right)^2 \quad \forall i \tag{17}$$

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i} \right); \quad \forall i$$

- Use $\sqrt{A_i + \epsilon}$ to avoid ill-conditioning.

RMSProp with Nesterov Momentum

- Possible to combine RMSProp with Nesterov Momentum

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right); \quad w_i \Leftarrow w_i + v_i \quad \forall i$$

- Maintenance of A_i is done with shifted gradients as well.

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right)^2 \quad \forall i \qquad (18)$$

AdaDelta and Adam

- Both methods derive intuition from RMSProp
  - AdaDelta track of an exponentially smoothed value of the *incremental changes* of weights Δw_i in previous iterations to decide parameter-specific learning rate.
  - Adam keeps track of exponentially smoothed gradients from previous iterations (in addition to normalizing like RMSProp).
- Adam is extremely popular method.

AdaDelta Optimizer

- Intuition: RMSProp variant

$$w_i \Leftarrow w_i - \underbrace{\frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

- α is replaced with a value (δ) that depends on the previous incremental updates. In each update, the value of Δw_i is the increment in the value of w_i.

$$\delta_i \Leftarrow \rho\delta_i + (1-\rho)(\Delta w_i)^2 \quad \forall i$$

- Update Rule:

$$w_i \Leftarrow w_i - \underbrace{\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

- Note: Adadelta update doesn't depend on learning rate(α)

Adam Optimizer

- Intuition: RMSProp variant

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i$$

- Exponential smoothing with ρ, ρf ∈ (0, 1)

$$F_i \Leftarrow \rho_f F_i + (1-\rho_f)\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i$$

- Update Rule: Below update is used at learning rate α_t in the t th iteration:

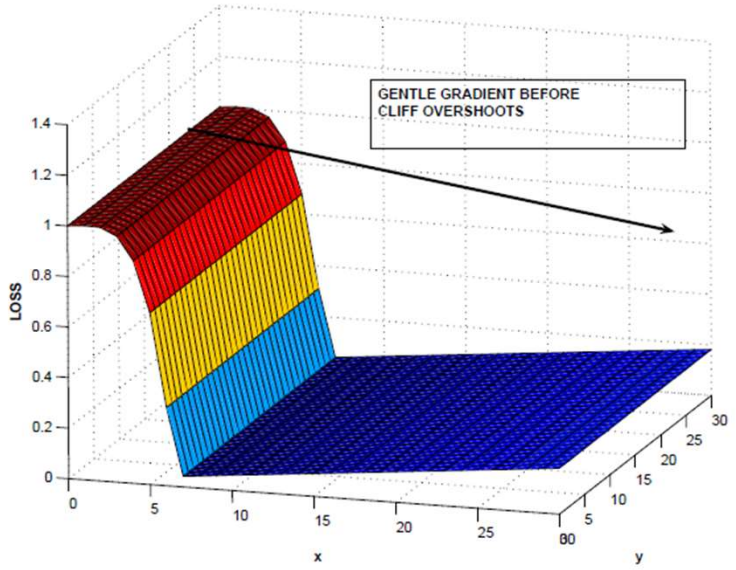
$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} F_i; \quad \forall i$$

$$\alpha_t = \alpha \underbrace{\left(\frac{\sqrt{1-\rho^t}}{1-\rho_f^t}\right)}_{\text{Adjust Bias}}$$

Second-order GD optimizer methods

- Conjugate Gradients and Hessian-Free Optimization
- Quasi-Newton Methods and BFGS

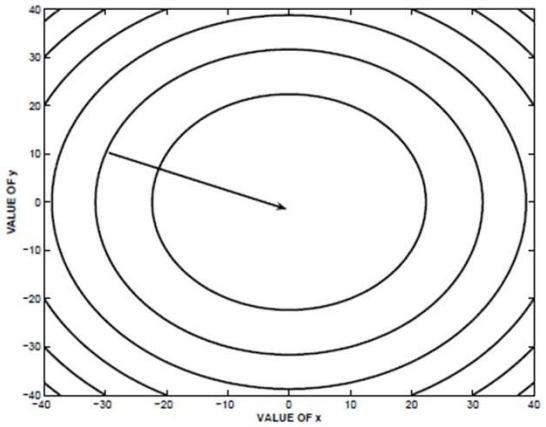
Why Second-Order Methods?



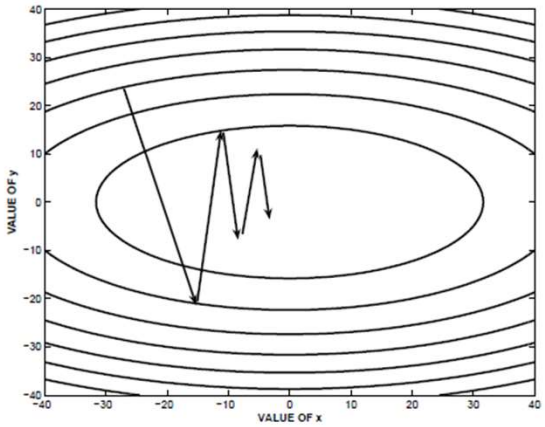
Small step-no overshoot
large step-may overshoot

- First-order methods are not enough when there is curvature.

Revisiting the Bowl



(a) Loss function is circular bowl
$L = x^2 + y^2$

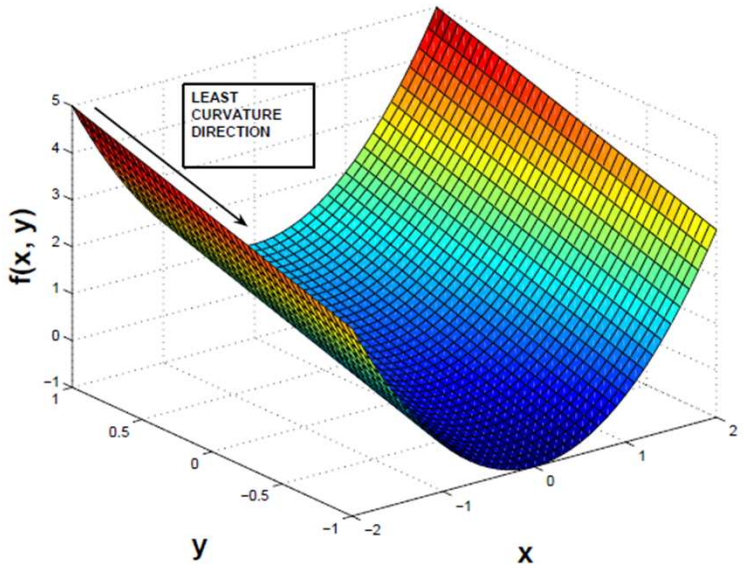


(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

different curvatures at axis

- High curvature directions cause bouncing in spite of higher gradient $\Rightarrow$ Need second-derivative for more information.

A Valley



- curvature along x is high
- curvature along y is low

- Gently sloping directions are better with less curvature!

The Hessian

- The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

- The partial derivatives use all pairwise parameters in the denominator.

- For a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}.

Quadratic Approximation of Loss Function

- One can write a quadratic approximation of the loss function with Taylor expansion about $\overline{W_0}$:

$$L(\overline{W}) \approx L(\overline{W}_0) + (\overline{W} - \overline{W}_0)^T [\nabla L(\overline{W}_0)] + \frac{1}{2}(\overline{W} - \overline{W}_0)^T H (\overline{W} - \overline{W}_0) \quad (19)$$

Zero-order term 1st order term 2nd order term

- One can derive a single-step optimality condition from initial point $\overline{W_0}$ by setting the gradient to 0.

Newton's Update

- Can solve quadratic approximation in one step from initial point $\overline{W_0}$.

  $\nabla L(\overline{W}) = 0$ [Gradient of Loss Function]
  $\nabla L(\overline{W_0}) + H(\overline{W} - \overline{W_0}) = 0$ [Gradient of Taylor approximation]

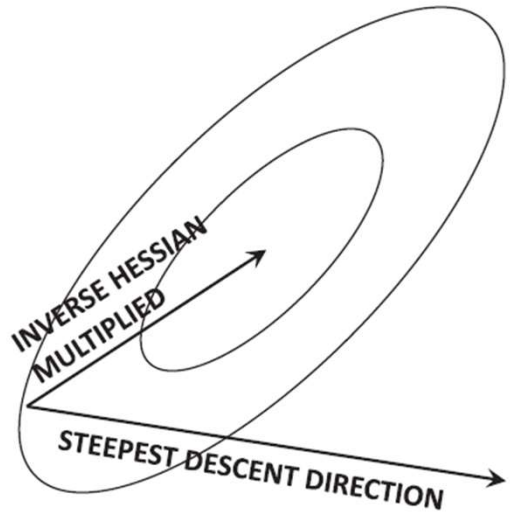
- Rearrange optimality condition to obtain Newton update:

$$\overline{W}^* \Leftarrow \overline{W_0} - H^{-1}[\nabla L(\overline{W_0})] \tag{20}$$

- Note the ratio of first-order to second-order $\Rightarrow$ Trade-off between speed and curvature

- Step-size not needed!

Why Second-Order Methods?



- Pre-multiplying with the inverse Hessian finds a trade-off between speed of descent and curvature.

Basic Second-Order Algorithm and Approximations

- Keep making Newton's updates to convergence (single step needed for quadratic function)
- Challenges
  - Even computing the Hessian is difficult!
  - Inverting it is even more difficult

- Solutions:
  - Approximate the Hessian.
  - Find an algorithm that works with projection $H\overline{v}$ for some direction $\overline{v}$.

Conjugate Gradient Method

- Get to optimal in d steps (instead of single Newton step) where d is number of parameters.

- Use optimal step-sizes to get best point along a direction.

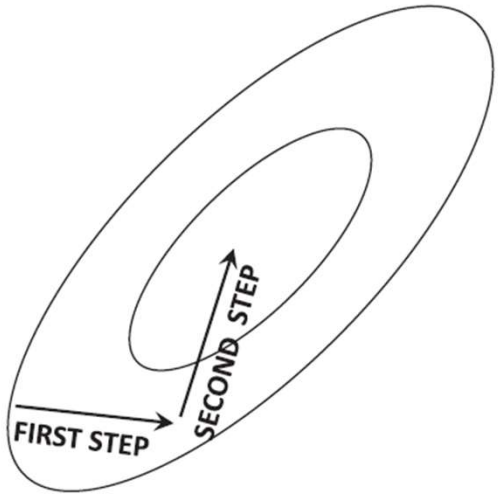
- *Thou shalt not worsen with respect to previous directions!*

- **Conjugate direction:** The gradient of the loss function on *any* point on an update direction is always orthogonal to the previous update directions.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t \qquad (21)$$

- For quadratic function, it requires d updates instead of single update of Newton method.

Conjugate Gradients on 2-Dimensional Quadratic



- Two conjugate directions are required to reach optimality

Conjugate Gradient Algorithm

- For quadratic functions only.
  - Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search.
  - Set $\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t$. Increment t by 1.
- For non-quadratic functions approximate loss function with Taylor expansion and perform $\ll d$ of the above steps. Then repeat.

Efficiently Computing Projection of Hessian

- The update requires computation of the *projection* of the Hessian rather than inversion of Hessian.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t \tag{22}$$

- Easy to perform numerically!

$$H\overline{v} \approx \frac{\nabla L(\overline{W}_0 + \delta\overline{v}) - \nabla L(\overline{W}_0)}{\delta} \tag{23}$$

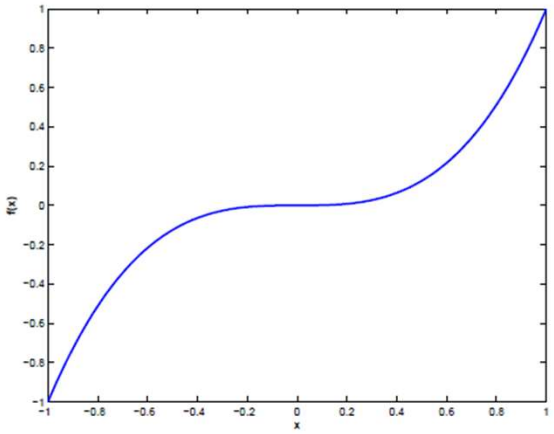
Other Second-Order Methods

- *Quasi-Newton Method:* A sequence of increasingly accurate approximations of the inverse Hessian matrix are used in various steps.

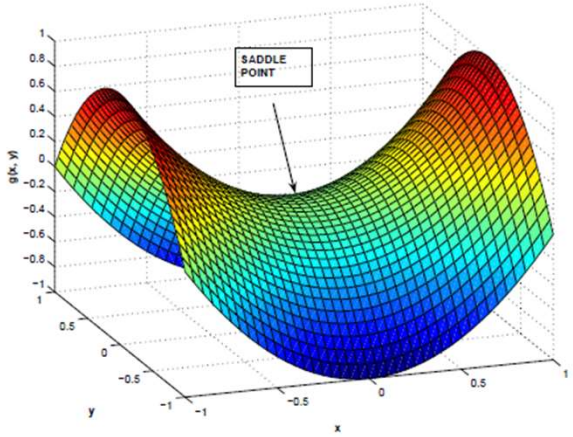
- Many variations of this approach.

- Commonly-used update is BFGS, which stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm and its limited memory variant L-BFGS.

Problems with Second-Order Methods



(a) $f(x) = x^3$
Degenerate



(b) $f(x) = x^2 - y^2$
Stationary

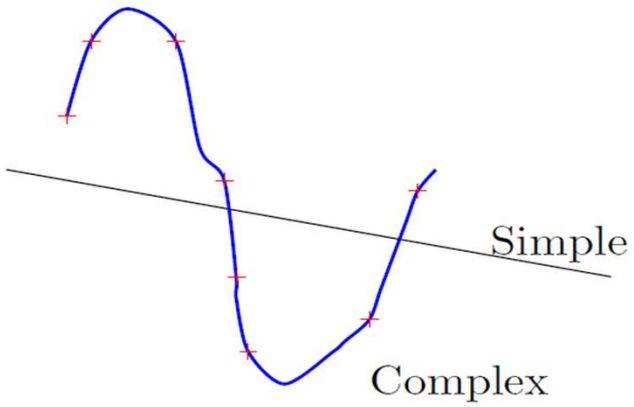
- Saddle points: Whether it is maximum or minimum depends on which direction we approach it from.

Bias and Variance

The points were drawn from a sinusoidal function (the true $f(x)$)

- Let us consider the problem of fitting a curve through a given set of points

Bias and Variance



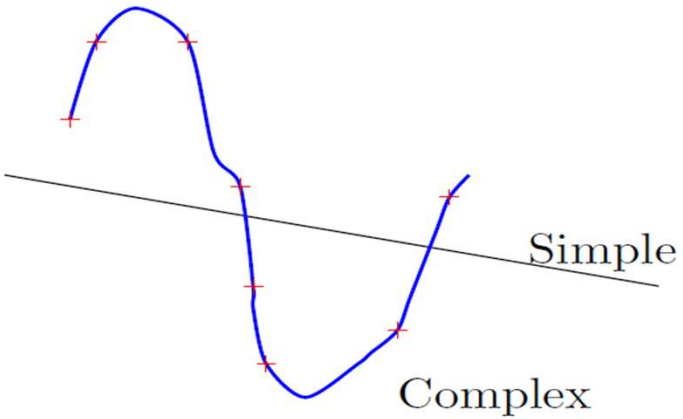
The points were drawn from a sinusoidal function (the true $f(x)$)

- Let us consider the problem of fitting a curve through a given set of points
- We consider two models :

$$\underset{(degree:1)}{Simple} \quad y = \hat{f}(x) = w_1 x + w_0$$

$$\underset{(degree:25)}{Complex} \quad y = \hat{f}(x) = \sum_{i=1}^{25} w_i x^i + w_0$$

Bias and Variance



The points were drawn from a sinusoidal function (the true $f(x)$)

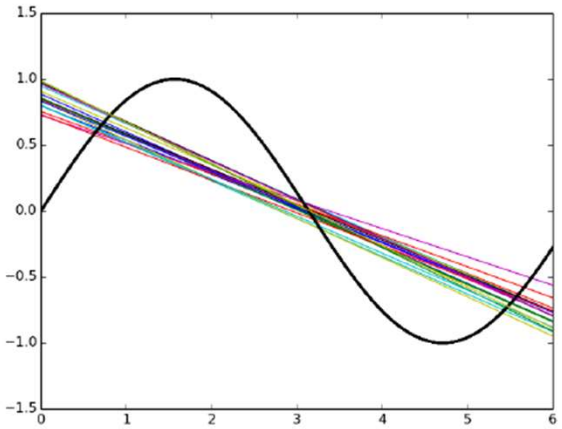
- Let us consider the problem of fitting a curve through a given set of points
- We consider two models :

$$\underset{(degree:1)}{Simple} \quad y = \hat{f}(x) = w_1 x + w_0$$

$$\underset{(degree:25)}{Complex} \quad y = \hat{f}(x) = \sum_{i=1}^{25} w_i x^i + w_0$$

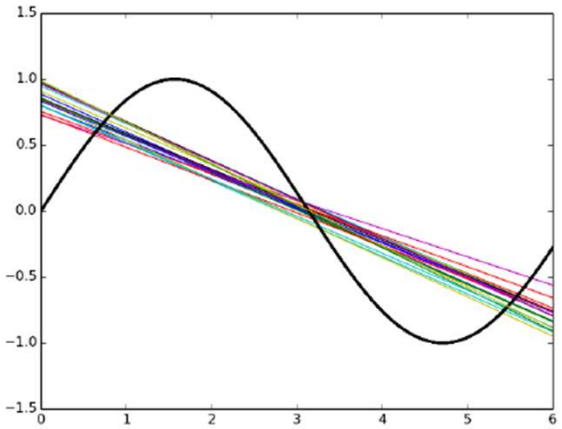
- Note that in both cases we are making an assumption about how y is related to x. We have no idea about the true relation $f(x)$
- The training data consists of 100 points

Bias and Variance



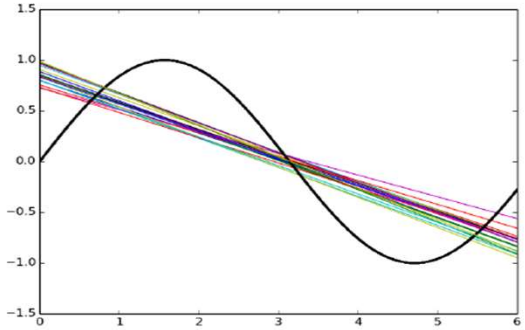
- Simple models trained on different samples of the data do not differ much from each other
- However they are very far from the true sinusoidal curve (under fitting)

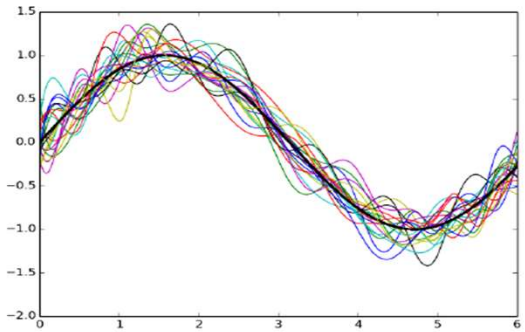
Bias and Variance



- Simple models trained on different samples of the data do not differ much from each other
- However they are very far from the true sinusoidal curve (under fitting)

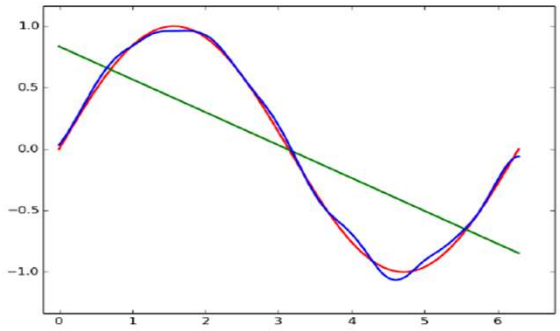
Bias and Variance





- Simple models trained on different samples of the data do not differ much from each other
- However they are very far from the true sinusoidal curve (under fitting)
- On the other hand, complex models trained on different samples of the data are very different from each other (high variance)

Bias



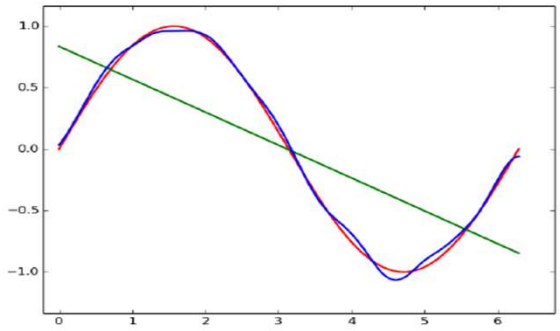
Green Line: Average value of $\hat{f}(x)$ for the simple model
Blue Curve: Average value of $\hat{f}(x)$ for the complex model
Red Curve: True model (f(x))

- Let $f(x)$ be the true model (sinusoidal in this case) and $\hat{f}(x)$ be our estimate of the model (simple or complex, in this case) then,

$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

- $E[\hat{f}(x)]$ is the average (or expected) value of the model

Bias



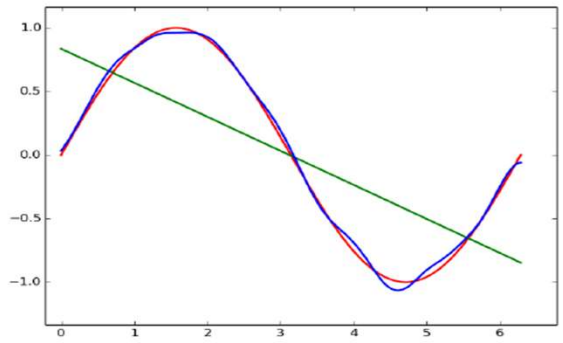
Green Line: Average value of $\hat{f}(x)$ for the simple model
Blue Curve: Average value of $\hat{f}(x)$ for the complex model
Red Curve: True model (f(x))

- Let $f(x)$ be the true model (sinusoidal in this case) and $\hat{f}(x)$ be our estimate of the model (simple or complex, in this case) then,

$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

- $E[\hat{f}(x)]$ is the average (or expected) value of the model

Bias



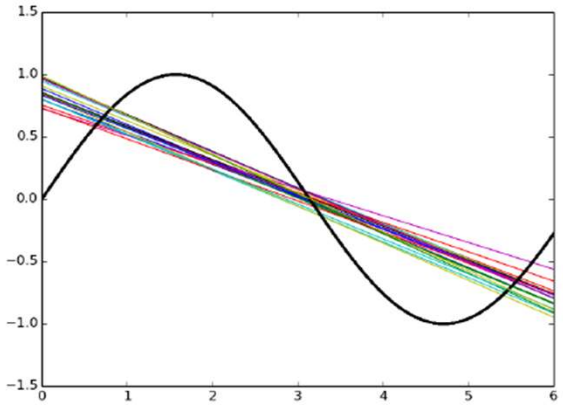
Green Line: Average value of $\hat{f}(x)$ for the simple model
Blue Curve: Average value of $\hat{f}(x)$ for the complex model
Red Curve: True model (f(x))

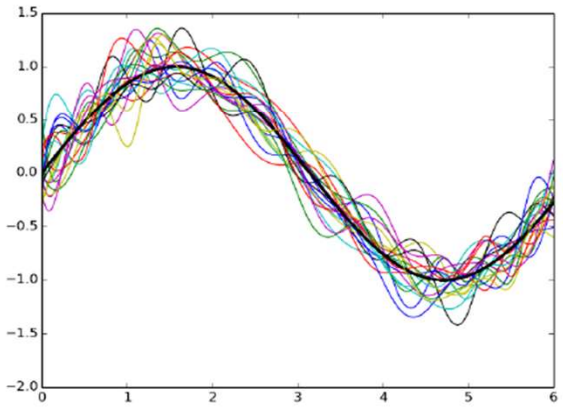
- Let $f(x)$ be the true model (sinusoidal in this case) and $\hat{f}(x)$ be our estimate of the model (simple or complex, in this case) then,

$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

- $E[\hat{f}(x)]$ is the average (or expected) value of the model
- We can see that for the simple model the average value (green line) is very far from the true value $f(x)$ (sinusoidal function)
- Mathematically, this means that the simple model has a high bias
- On the other hand, the complex model has a low bias

Variance





- We now define,

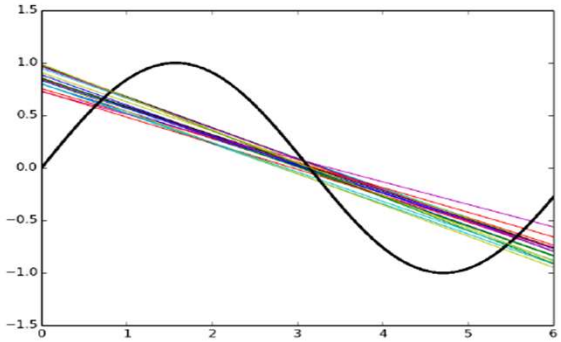
$$\text{Variance } (\hat{f}(x)) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

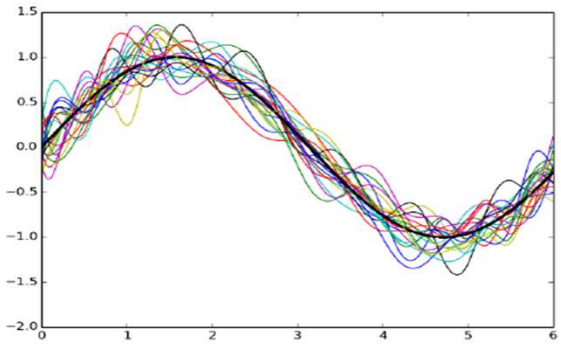
(Standard definition from statistics)

- Roughly speaking it tells us how much the different $\hat{f}(x)$'s (trained on different samples of the data) differ from each other

- It is clear that the simple model has a low variance whereas the complex model has a high variance

Bias and Variance





- In summary (informally)
- Simple model: high bias, low variance
- Complex model: low bias, high variance
- There is always a trade-off between the bias and variance
- Both bias and variance contribute to the mean square error. Let us see how

Bias-Variance Equation

- Let $E[MSE]$ be the expected mean-squared error of the fixed set of test instances over different samples of training data sets.

$$E[MSE] = \text{Bias}^2 + \text{Variance} + \text{Noise} \quad (1)$$

  - In linear models, the bias component will contribute more to $E[MSE]$.

  - In polynomial models, the variance component will contribute more to $E[MSE]$.

- We have a trade-off, when it comes to choosing model complexity!

Noise Component

- Unlike bias and variance, noise is a property of the *data* rather than the model.

- Noise refers to unexplained variations ϵ_i of data from true model $y_i = f(x_i) + \epsilon_i$.

- Real-world examples:

  - Human mislabeling of test instance $\Rightarrow$ Ideal model will never predict it accurately.

  - Error during collection of temperature due to sensor malfunctioning.

- Cannot do anything about it even if seeded with knowledge about true model.

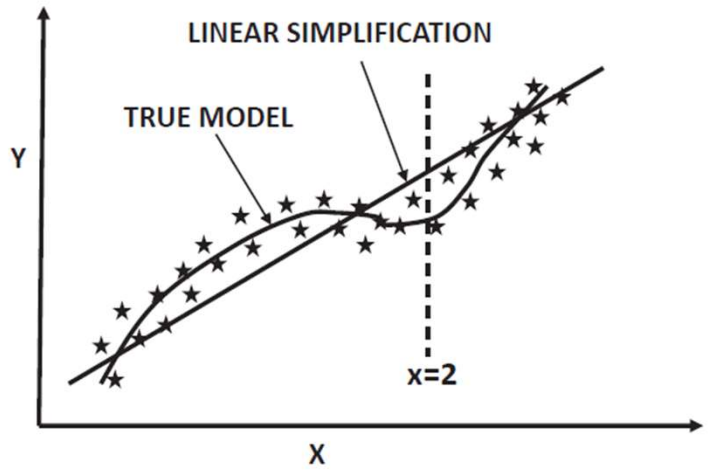
Memorization vs Generalization

- Why is the accuracy on seen data higher?
  - Trained model remembers some of the irrelevant nuances.
- When is the gap between seen and unseen accuracy likely to be high?
  - When the amount of data is limited.
  - When the model is complex (which has higher *capacity* to remember nuances).
  - The combination of the two is a deadly cocktail.
- A high accuracy gap between the predictions on seen and unseen data is referred to as *overfitting*.

What is Model Generalization?

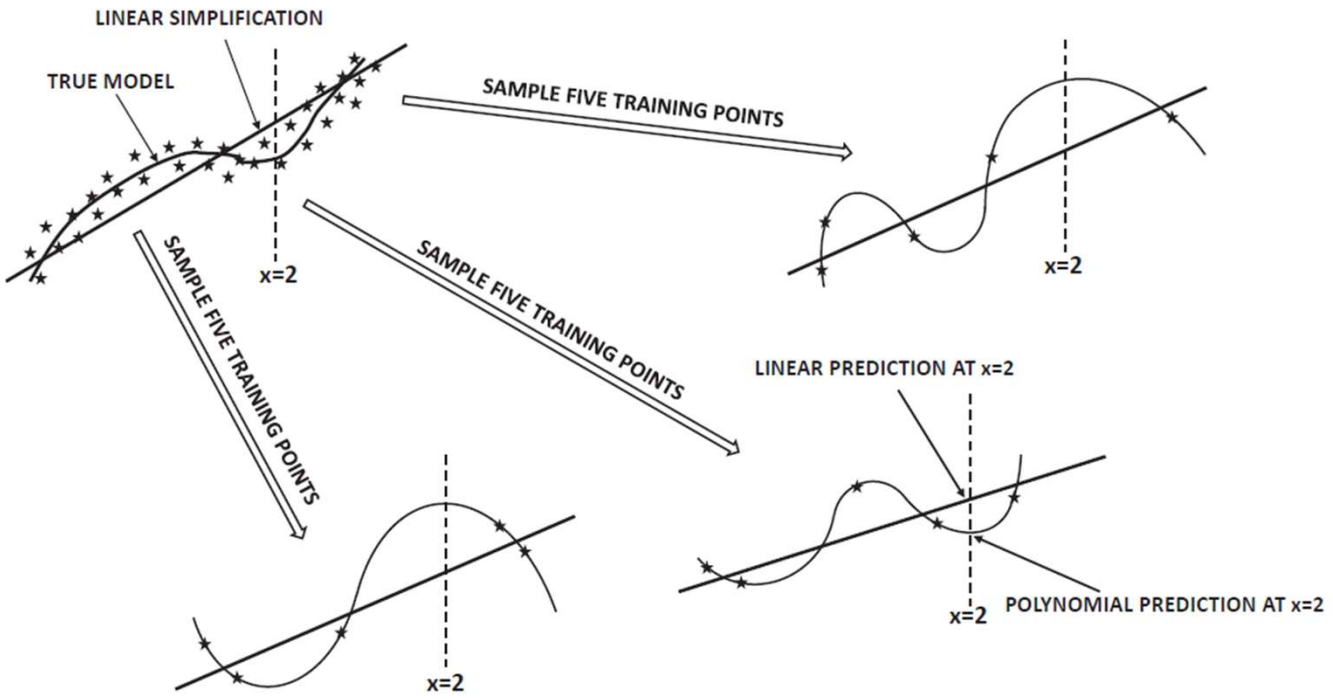
- In a machine learning problem, we try to generalize the known dependent variable on seen instances to unseen instances.
  - Unseen $\Rightarrow$ The model did not see it during training.
  - Given training images with seen labels, try to label an unseen image.
  - Given training emails labeled as spam or nonspam, try to label an unseen email.
- The classification accuracy on instances used to train a model is usually higher than on unseen instances.
  - We only care about the accuracy on unseen data.

Example: Predict y from x



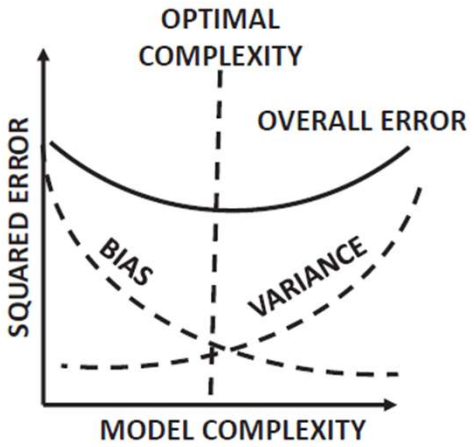
- **First impression:** Polynomial model such as $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$ is "better" than linear model $y = w_0 + w_1 x$.
  - Bias-variance trade-off says: "Not necessarily! How much data do you have?"

Different Training Data Sets with Five Points



- Zero error on training data but wildly varying predictions of $x = 2$

The Bias-Variance Trade-Off



- Optimal point of model complexity is somewhere in middle.

Key Takeaway of Bias-Variance Trade-Off

- A model with greater complexity might be *theoretically* more accurate (i.e., low bias).
  - But you have less control on what it might predict on a tiny training data set.
  - Different training data sets will result in widely *varying* predictions of same test instance.
  - Some of these must be wrong $\Rightarrow$ Contribution of model variance.
- *A more accurate model for infinite data is not a more accurate model for finite data.*

The Bias and Variance Trade-off

- Why do we care about this bias variance tradeoff and model complexity?

- Deep Neural networks are highly complex models.
- Many parameters, many non-linearities.
- It is easy for them to overfit and drive training error to 0.
- Hence we need some form of regularization.

How to Detect Overfitting

- The error on test data might be caused by several reasons.
  - Other reasons might be bias (underfitting), noise, and poor convergence.
- Overfitting shows up as a large gap between in-sample and out-of-sample accuracy.
- First solution is to collect more data.
  - More data might not always be available!

Different forms of regularization

- l_2 regularization
- Dataset augmentation
- Parameter Sharing and tying
- Adding Noise to the inputs
- Adding Noise to the outputs
- Early stopping
- Ensemble methods
- Dropout

Generalization Issues in Model Tuning and Evaluation

- **Model tuning and Hyperparameter choice:**
  - If one tuned the neural network with the same data that were used to train it, one would not obtain very good results because of overfitting.
  - The hyperparameters (e.g., regularization parameter) are tuned on a separate held-out set than the one on which the weight parameters on the neural network are learned.

- **Issues with Training at large scale for different hyperparameter settings:**
  - Common strategy is to run the training process of each setting for a fixed number of epochs.
  - Multiple runs are executed over different choices of hyperparameters in different threads of execution.
- **How to Detect Need to collect more data:**
  - **High bias=>Underfitting and High Variance=>Overfitting**
  - With increased training data, the training accuracy will reduce, whereas the test/validation accuracy will increase.

Generalization Issues in Model Tuning and Evaluation

- A given data set should always be divided into three parts defined according to the way in which the data are used:

1. **Training data:** part of data used to build training model
2. **Validation data:** part of the data used for model tuning
   - Strictly speaking, the validation data is also a part of the training data, because it influences the final model
3. **Testing data:** part of the data used to test the accuracy of the final (tuned) model.
   - It is important that the testing data are not even looked at during the process of parameter tuning and model selection to prevent overfitting.
   - The testing data are used only once at the very end of the process.

Generalization Issues in Model Tuning and Evaluation

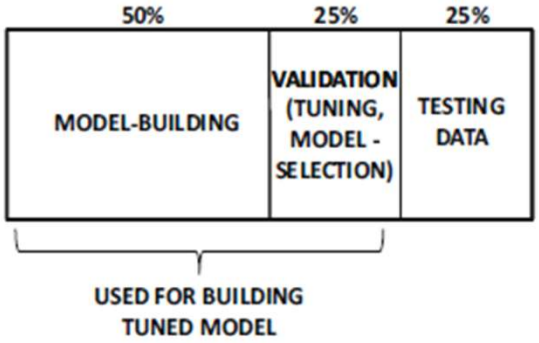


Figure 4.4: Partitioning a labeled data set for evaluation design

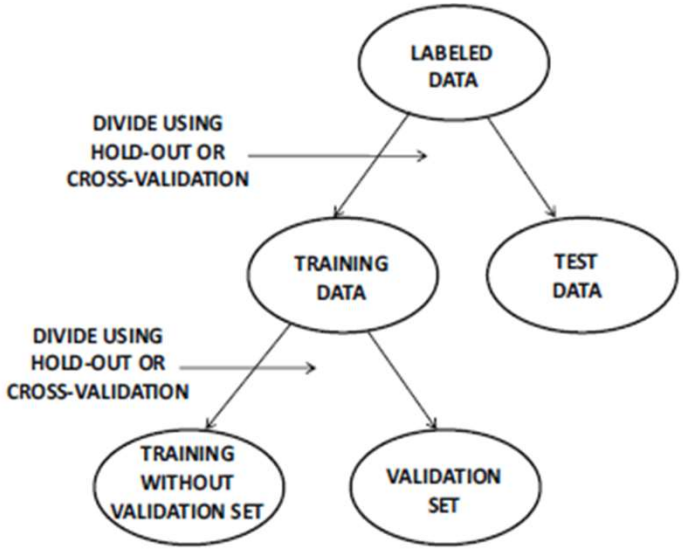


Figure 4.5: Hierarchical division into training, validation, and testing portions

Holdout

- In the hold-out method, a fraction of the instances are used to build the training model.
- The remaining instances, which are also referred to as **the held-out instances, are used for testing.**
- The accuracy of predicting the labels of the held-out instances is then reported as the overall accuracy.

Holdout

- Pros:
  - Such an approach ensures that the reported accuracy is not a result of overfitting to the specific data set, because different instances are used for training and testing.
  - Simple and efficient
- Cons:
  - underestimates the true accuracy
  - pessimistic bias in evaluation due to class imbalance

Cross Validation

- In the cross-validation method, the labeled data is divided into *q* equal segments.
- One of the *q* segments is used for testing, and the remaining *(q−1)* segments are used for training.
- This process is repeated *q* times by using each of the *q* segments as the test set.
- The average accuracy over the *q* different test sets is reported.

Ensemble Methods

Ensemble Methods

- Ensemble methods derive their inspiration from the bias-variance trade-off.
- One way of reducing the error of a classifier is to find a way to reduce either its bias or the variance without affecting the other component.
- Ensemble methods are used commonly in machine learning,
  - Examples:
    - Bagging ----------variance reduction
    - Boosting ---------bias reduction.

    Boosting Reference

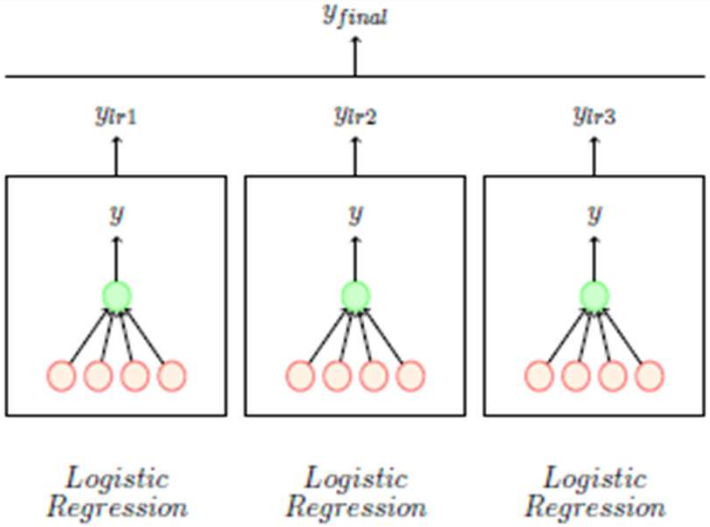
Ensemble Methods

- Most ensemble methods in neural networks are focused on variance reduction.
  - as neural networks are valued for their ability to build arbitrarily complex models in which the bias is relatively low.
  - However, operating at the complex end of the bias variance trade-off almost always leads to higher variance, which is manifested as overfitting.
- Therefore, the goal of most ensemble methods in the neural network setting is variance reduction (i.e., better generalization).

Ensemble Methods: Bagging

- In bagging, the training data is sampled with replacement.
- The sample size *s* may be different from the size of the training data size *n*, although it is common to set s to *n*
- The resampled data will contain duplicates, and about a fraction $(1-1/n)^n \approx 1/e$ of the original data set will not be included at all. Here, the notation *e* denotes the base of the natural logarithm.
- A model is constructed on the resampled training data set, and each test instance is predicted with the resampled data.
- The entire process of resampling and model building is repeated *m* times

Ensemble Methods: Bagging



Each model trained with a different sample of the data (sampling with replacement)

- Bagging: form an ensemble using different instances of the same classifier
- From a given dataset, construct multiple training sets by sampling with replacement $(T_1, T_2, ..., T_k)$
- Train i^{th} instance of the classifier using training set T_i

Ensemble Methods: Bagging

- For a given test instance, each of these *m* models is applied to the test data.
- The predictions from different models are then averaged to yield a single robust prediction.
- In bagging, the best results are often obtained by choosing values of $s <<< n$.

Challenges:

- The main challenge in directly using bagging for neural networks is that one must construct multiple training models, which is highly inefficient unless training is on multiple GPU processors.

Ensemble Methods: Subsampling

- Subsampling is similar to bagging, except that the different models are constructed on the samples of the data created **without replacement.**
- The predictions from the different models are averaged.
- In this case, it is essential to choose $s < n$

Ensemble Methods: Parametric Model Selection and Averaging

- The presence of a large number of hyper parameters creates problems in model construction, because the performance might be sensitive to the particular configuration used.
- One possibility is to hold out a portion of the training data and try different combinations of parameters and model choices.
- The selection that provides the highest accuracy on the held-out portion of the training data is then used for prediction.
- This is the standard approach used for parameter tuning in all machine learning models, and is referred to as **model selection/bucket-of-models**.

Ensemble Methods: Randomized Connection Dropping

- The random dropping of connections between different layers in a multilayer neural network often leads to diverse models in which different combinations of features are used to construct the hidden variables.
- The dropping of connections between layers does tend to create less powerful models because of the addition of constraints to the model-building process.
  - However, since different random connections are dropped from different models, the predictions from different models are very diverse.
  - The averaged prediction from these different models is often highly accurate.

Note: The weights of different models are not shared in this approach