Neural Networks

Lecture 6: Optimization & regularization
Cost function \[ J(\theta) \]

Gradient
\[ g = \nabla_{\theta} J(\theta) \]
\[ g_i = \frac{\partial}{\partial \theta_i} J(\theta) \]

\[ \theta \leftarrow \theta - \epsilon g \]
<table>
<thead>
<tr>
<th>Lesson</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Probability and Information Theory</td>
</tr>
<tr>
<td>3</td>
<td>Basics of Machine Learning</td>
</tr>
<tr>
<td>4</td>
<td>Deep Feed-forward networks</td>
</tr>
<tr>
<td>5</td>
<td>Back-propagation</td>
</tr>
<tr>
<td>6</td>
<td>Optimization and regularization</td>
</tr>
<tr>
<td>7</td>
<td>Convolutional Networks</td>
</tr>
<tr>
<td>8</td>
<td>Sequence Modeling: recurrent networks</td>
</tr>
<tr>
<td>9</td>
<td>Applications</td>
</tr>
<tr>
<td>10</td>
<td>Software and Practical methods</td>
</tr>
<tr>
<td>11</td>
<td>Representational Learning</td>
</tr>
<tr>
<td>12</td>
<td>Deep Generative Models</td>
</tr>
<tr>
<td>13</td>
<td>Deep Reinforcement Learning</td>
</tr>
<tr>
<td>14</td>
<td>Deep Learning and the Brain</td>
</tr>
<tr>
<td>15</td>
<td>Conclusions and Future Perspective</td>
</tr>
<tr>
<td>16</td>
<td>Project presentations</td>
</tr>
</tbody>
</table>
Cost function: $J(\theta)$

Gradient:

$$g = \nabla_{\theta} J(\theta)$$
$$g_i = \frac{\partial}{\partial \theta_i} J(\theta)$$

$$\theta \leftarrow \theta - \epsilon g$$
Learning objectives

• Understand the main forms of regularization used in practical neural networks

• Understand stochastic gradient descent and some variants
Regularization methods

Optimization algorithms
Model generalization

**Underfitting:** model is not able to obtain sufficiently low error in the training set

**Overfitting:** gap between training and test error is too large

**Capacity:** ability to fit a wide variety of functions
Model generalization

Graph showing the relationship between error and capacity, with zones for underfitting and overfitting.
Model generalization

We can control the model capacity by adding or removing functions in the hypothesis space but also by giving preference to certain functions.

**Regularization:** modification of algorithm intended to decrease the generalization error but not the training error.

\[ J(w) = \text{MSE}_{\text{train}} + \lambda w^T w \]
L2 regularization

\[ \tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta) \]

\[ \tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y) \]  

Weight decay

Weights, not biases!
L2 regularization

\[ \tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta) \]

\[ \tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y) \]

Weight decay
Weights, not biases!

Ridge regression,
Tikhonov regularization
L2 regularization

\[ \tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta) \]

\[ \tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y) \]

Weight decay
Ridge regression, Tikhonov regularization
Weights, not biases!

\[ \nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y) \]

\[ w \leftarrow w - \epsilon (\alpha w + \nabla_w J(w; X, y)) \]

\[ w \leftarrow (1 - \epsilon \alpha) w - \epsilon \nabla_w J(w; X, y) \]
CHAPTER 7. REGULARIZATION FOR DEEP LEARNING

Figure 7.1: An illustration of the effect of $L_2$ (or weight decay) regularization on the value of the optimal $w$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the $L_2$ regularizer. At $\tilde{w}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of $J$ is small. The objective function does not increase much when moving horizontally away from $w^\ast$. Because the objective function does not express as strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls $w_1$ close to zero. In the second dimension, the objective function is very sensitive to movements away from $w^\ast$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of $w_2$ relatively little.

Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact. In directions that do not contribute to reducing the objective function, a small eigenvalue of the Hessian tells us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training.

So far we have discussed weight decay in terms of its effect on the optimization of an abstract, general, quadratic cost function. How do these effects relate to machine learning in particular? We can find out by studying linear regression, a model for which the true cost function is quadratic and therefore amenable to the same kind of analysis we have used so far. Applying the analysis again, we will be able to obtain a special case of the same results, but with the solution now phrased in terms of the training data. For linear regression, the cost function is...
**L1 regularization**

\[ \tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta) \quad \Omega(\theta) = ||w||_1 = \sum_i |w_i| \]

\[ \tilde{J}(w; X, y) = \alpha ||w||_1 + J(w; X, y) \quad \text{LASSO} \]

\[ \nabla_w \tilde{J}(w; X, y) = \alpha \text{sign}(w) + \nabla_w J(X, y; w) \]
Norm Penalties

• L1: Encourages sparsity, equivalent to MAP Bayesian estimation with Laplace prior

• Squared L2: Encourages small weights, equivalent to MAP Bayesian estimation with Gaussian prior
Dataset Augmentation

Affine Distortion → Horizontal flip
Noise → Random Translation
Elastic Deformation → Hue Shift

(Goodfellow 2016)
Multi-Task Learning

The model can generally be divided into two kinds of parts and associated parameters:

1. Task-specific parameters (which only benefit from the examples of their task to achieve good generalization). These are the upper layers of the neural network in figure 7.2.

2. Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). These are the lower layers of the neural network in figure 7.2.

Figure 7.2: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $h^{(1)}$ and $h^{(2)}$) can be learned to yield a shared representation $h^{(\text{shared})}$. The underlying assumption is that there exists a common pool of factors that explain the variations in the input $x$, while each task is associated with a subset of these factors. In this example, it is additionally assumed that top-level hidden units $h^{(1)}$ and $h^{(2)}$ are specialized to each task (respectively predicting $y^{(1)}$ and $y^{(2)}$) while some intermediate-level representation $h^{(\text{shared})}$ is shared across all tasks. In the unsupervised learning context, it makes sense for some of the top-level factors to be associated with none of the output tasks ($h^{(3)}$): these are the factors that explain some of the input variations but are not relevant for predicting $y^{(1)}$ or $y^{(2)}$.

Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be...
CHAPTER 7. REGULARIZATION FOR DEEP LEARNING

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or epochs). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

From the point of view of deep learning, the underlying prior belief is the following: among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See figure 7.3 for an example of this behavior. This behavior occurs very reliably. This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.
Early stopping

terminate while validation set performance is better

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or epochs). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See figure 7.3 for an example of this behavior. This behavior occurs very reliably.

This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.
Early Stopping and Weight Decay

Figure 7.4

- Left: An illustration of the effect of early stopping. The solid contour lines indicate the contours of the negative log-likelihood. The dashed line indicates the trajectory taken by SGD beginning from the origin. Rather than stopping at the point $w^\ast$ that minimizes the cost, early stopping results in the trajectory stopping at an earlier point $\tilde{w}$.

- Right: An illustration of the effect of $L_2$ regularization for comparison. The dashed circles indicate the contours of the $L_2$ penalty, which causes the minimum of the total cost to lie nearer the origin than the minimum of the unregularized cost.

We are going to study the trajectory followed by the parameter vector during training. For simplicity, let us set the initial parameter vector to the origin, that is $w(0) = 0$. Let us study the approximate behavior of gradient descent on $J$ by analyzing gradient descent on $\hat{J}$:

$$w(\tau) = w(\tau - 1) - \varepsilon \nabla \hat{J}(w(\tau - 1))$$

(7.35)

$$= w(\tau - 1) - \varepsilon H(w(\tau - 1))w^\ast$$

(7.36)

Let us now rewrite this expression in the space of the eigenvectors of $H$, exploiting the eigendecomposition of $H$:

$$H = Q\Lambda Q^\top$$

where $\Lambda$ is a diagonal matrix and $Q$ is an orthonormal basis of eigenvectors.

$$w(\tau) = (I - \varepsilon \Lambda)w(\tau - 1)w^\ast$$

(7.37)

For neural networks, to obtain symmetry breaking between hidden units, we cannot initialize all the parameters to 0, as discussed in section 6.2. However, the argument holds for any other initial value $w(0)$.
Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an 8 detector on the dataset depicted above, containing an 8, a 6, and a 9. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the 9 and repeats the 8. On this dataset, the detector learns that a loop on top of the digit corresponds to an 8. On the second dataset, we repeat the 9 and omit the 6. In this case, the detector learns that a loop on the bottom of the digit corresponds to an 8. Each of these individual classification rules is brittle, but if we average their output then the detector is robust, achieving maximal confidence only when both loops of the 8 are present.

Bagging is a method that allows the same kind of model, training algorithm, and objective function to be reused several times. Specifically, bagging involves constructing $k$ different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the original dataset and also contains several duplicate examples (on average around $2/3$ of the examples from the original dataset are found in the resulting training set, if it has the same size as the original). Model $i$ is then trained on dataset $i$. The differences between which examples are included in each dataset result in differences between the trained models. See figure 7.5 for an example.

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all of the models are trained on the same dataset. Differences in random initialization, random selection of minibatches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the
Bagging

Bootstrap aggregating

Reduce generalization error by combining several models

Train different models (k different datasets) and then vote

\( k \) regression models, each making error \( \epsilon_i \)

\[ \mathbb{E}[\epsilon_i^2] = \nu \]

\[ \mathbb{E}[\epsilon_i \epsilon_j] = c \]

\[
\frac{1}{k} \sum_i \epsilon_i \quad \mathbb{E} \left[ \left( \frac{1}{k} \sum_i \epsilon_i \right)^2 \right] = \frac{1}{k^2} \mathbb{E} \left[ \sum_i \left( \epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j \right) \right] \\
= \frac{1}{k} \nu + \frac{k-1}{k} c.
\]
Ensemble methods are impractical for large neural networks

It is common to use 5-10 neural networks
(6 to win ILSVRC 2014)

Dropout trains the ensemble of subnetworks formed by removing non-output units from an underlying base network
Dropout trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network. Here, we begin with a base network with two visible units and two hidden units. There are sixteen possible subsets of these four units. We show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network. In this small example, a large proportion of the results in networks have no input units or no paths connecting the input to the output. This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.
Dropout

Remove a unit = multiply its output by 0  (Binary mask)
Dropout

Remove a unit = multiply its output by 0  (Binary mask)

$P_{\text{keep}}=0.5$ hidden nodes

$P_{\text{keep}}=0.8$ input nodes
Dropout

Remove a unit = multiply its output by 0  

\[ P_{\text{keep}} = 0.5 \text{ hidden nodes} \]

\[ P_{\text{keep}} = 0.8 \text{ input nodes} \]

Parameter sharing

Each model is trained for one sample

(Binary mask)
Ensemble predictor: use the full underlying network with weights are scaled to maintain same input to a node during training and testing (weight scaling inference rule)
Regularization methods

Optimization algorithms
Cost function

\[ J(\theta) \]

Gradient

\[ g = \nabla_{\theta} J(\theta) \]
\[ g_i = \frac{\partial}{\partial \theta_i} J(\theta) \]

\[ \theta \leftarrow \theta - \epsilon g \]
Learning & optimization

What we have

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

Empirical risk

What we want

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

Risk

Empirical risk optimization:

Some loss functions cannot be optimised efficiently

E.g. 0-1 loss function  \rightarrow  negative log likelihood of correct class
Learning & optimization

What we have

\[ J(\theta) = \mathbb{E}_{(x,y) \sim \hat{p}_{\text{data}}} L(f(x; \theta), y) \]

Empirical risk

What we want

\[ J^*(\theta) = \mathbb{E}_{(x,y) \sim p_{\text{data}}} L(f(x; \theta), y) \]

Risk

Empirical risk optimization:

Some loss functions cannot be optimised efficiently

E.g. 0-1 loss function  negative log likelihood of correct class

Prone to overfitting
CHAPTER 7. REGULARIZATION FOR DEEP LEARNING

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or epochs). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

From the point of view of deep learning, the underlying prior belief is the following: among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks. Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See figure 7.3 for an example of this behavior. This behavior occurs very reliably. This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.
Stochastic methods

Objective function decomposes as a sum over the training samples

\[ J(\theta) = \mathbb{E}_{(x,y) \sim \hat{p}_{\text{data}}} L(f(x; \theta), y) \]

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

Update of parameters based on an expected value of the cost function estimated using only a subset of the terms
Stochastic methods

Maximum likelihood estimator:

\[ \theta_{ML} = \arg \max_{\theta} \sum_{i=1}^{m} \log p_{\text{model}}(x^{(i)}, y^{(i)}; \theta) \]

\[ J(\theta) = \mathbb{E}_{x, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(x, y; \theta) \]
Stochastic methods

Maximum likelihood estimator:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)}, y^{(i)}; \theta) \quad J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(x, y; \theta)$$

Most of properties that we need from the cost functions are expectations over the training set

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{data}} \nabla_{\theta} \log p_{model}(x, y; \theta)$$
Stochastic methods

Maximum likelihood estimator:

\[ \theta_{\text{ML}} = \arg \max_{\theta} \sum_{i=1}^{m} \log p_{\text{model}}(x^{(i)}, y^{(i)}; \theta) \]

\[ J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(x, y; \theta) \]

Most of properties that we need from the cost functions are expectations over the training set

\[ \nabla_{\theta} J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{model}}(x, y; \theta) \]

Estimate expectations by randomly taking small number of examples from dataset and then averaging

E.g. standard error of the mean (n samples) \[ \sigma / \sqrt{n} \]

n = 100 \times 100 \quad n = 10000 \quad \sigma / \sqrt{n} \times 1/10
Stochastic methods

Optimization methods that use a subset of training data (minibatch)

Stochastic gradient descent (SGD)

Larger batches more accurate gradients but with less than linear returns

Sizes: 16-256

Selected randomly

Epoch: a full pass through the training set
Stochastic gradient descent

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\theta$

$k \leftarrow 1$

while stopping criterion not met do

Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Apply update: $\theta \leftarrow \theta - \epsilon_k \hat{g}$

$k \leftarrow k + 1$

end while
Momentum
\[ v \leftarrow \alpha v - \epsilon \nabla_\theta \left( \frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \right) \]

\[ \theta \leftarrow \theta + v \]
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate $\epsilon$, momentum parameter $\alpha$

Require: Initial parameter $\theta$, initial velocity $v$

while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient estimate: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$.
    Compute velocity update: $v \leftarrow \alpha v - \epsilon g$.
    Apply update: $\theta \leftarrow \theta + v$.
end while
Parameter initialization

Initial point from which to begin the iterations

It can determine whether algorithm converges or fails
Parameter initialization

Initial point from which to begin the iterations

It can determine whether algorithm converges or fails

It needs to break symmetry between different units
Parameter initialization

For a fully connected layer with $m$ inputs and $n$ outputs:

**Simple initialization**

$$U\left( -\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}} \right)$$

**Glorot initialization**

$$W_{i,j} \sim U \left( -\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}} \right)$$

Biases:

- Output units to match the marginal probabilities of targets
- Avoid saturation (bias for ReLU @ 0.1)
Adaptive learning rates

One of the most difficult to set hyperparameters because it significantly affects model performance

Cost function sensitive to some directions of parameters space and insensitive to others

Separate learning rate for each parameter and adaptive throughout the training

E.g. delta-bar-delta algorithm

If gradient remains the same sign: rate increase
If gradient changes sign: rate decreases
AdaGrad

\[ \theta_t = \theta_{t-1} - \alpha g_t \]

Adapts learning rate of all model parameters inversely to the historical values of gradients:

- parameters with largest gradients have a rapid decrease
- parameters with small gradients have a relatively small decrease
Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate $\epsilon$
Require: Initial parameter $\theta$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability

Initialize gradient accumulation variable $r = 0$

while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_{i} L(f(x^{(i)}; \theta), y^{(i)})$.
    Accumulate squared gradient: $r \leftarrow r + g \odot g$.
    Compute update: $\Delta \theta \leftarrow \frac{\epsilon}{\delta + \sqrt{r}} \odot g$. (Division and square root applied element-wise)
    Apply update: $\theta \leftarrow \theta + \Delta \theta$.
end while
AdaGrad is designed to converge rapidly for convex problems.

For NN learning trajectory can pass through many structures before arriving at a convex region.

RMSprop discards history from remote past so that it can converge rapidly after finding a convex region.
Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate $\epsilon$, decay rate $\rho$

Require: Initial parameter $\theta$

Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small numbers

Initialize accumulation variables $r = 0$

while stopping criterion not met do

Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta); y^{(i)})$.

Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$.

Compute parameter update: $\Delta \theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$. $(\frac{1}{\sqrt{\delta + r}}$ applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta \theta$.

end while
Adam

Adam: adaptive moments

Adam = RMSprop + momentum
Second order methods

Cost function

$J(\theta)$

Gradient

$g = \nabla_{\theta} J(\theta)$

$g_i = \frac{\partial}{\partial \theta_i} J(\theta)$

Hessian

$H_{i,j} = \frac{\partial}{\partial \theta_j} g_i$

(Goodfellow 2015)
Taylor series approximation

\[ f(x) = f(x_0) + (x-x_0)f'(x) + \frac{1}{2}(x-x_0)^2 f''(x) + \ldots \]

\[ J(\theta) = J(\theta_0) + (\theta - \theta_0)^\top g + \frac{1}{2}(\theta - \theta_0)^\top H(\theta - \theta_0) + \ldots \]

Baseline
Linear change due to gradient
Correction from directional curvature

(Goodfellow 2015)
Newton method

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_{\theta} J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta - \theta_0). \]

\[ \theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0). \]

Very expensive

Attracted to saddle points
The old view of SGD as difficult

- SGD usually moves downhill
- SGD eventually encounters a critical point
- Usually this is a minimum
- However, it is a local minimum
- J has a high value at this critical point
- Some global minimum is the real target, and has a much lower value of J
The new view:

- SGD usually moves downhill
- SGD eventually encounters a critical point
- Usually this is a saddle point

Newton’s method gets stuck at critical points but this does not happen very much to SGD

- So most local minima have low J!
- Major implication: most minima are good, and this is more true for big models.
Ideally, we would like to arrive at the global minimum, but this might not be possible. This local minimum performs nearly as well as the global one, so it is an acceptable halting point.

This local minimum performs poorly, and should be avoided.

Why convergence may not happen

- Never stop if function doesn’t have a local minimum
- Get “stuck,” possibly still moving but not improving
  - Too bad of conditioning
  - Too much gradient noise
  - Overfitting
  - Other?

- Usually we get “stuck” before finding a critical point
- Only Newton’s method and related techniques are attracted to saddle points
<table>
<thead>
<tr>
<th>Lesson</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Probability and Information Theory</td>
</tr>
<tr>
<td>3</td>
<td>Basics of Machine Learning</td>
</tr>
<tr>
<td>4</td>
<td>Deep Feed-forward networks</td>
</tr>
<tr>
<td>5</td>
<td>Back-propagation</td>
</tr>
<tr>
<td>6</td>
<td>Optimization and regularization</td>
</tr>
<tr>
<td>7</td>
<td>Convolutional Networks</td>
</tr>
<tr>
<td>8</td>
<td>Sequence Modeling: recurrent networks</td>
</tr>
<tr>
<td>9</td>
<td>Applications</td>
</tr>
<tr>
<td>10</td>
<td>Software and Practical methods</td>
</tr>
<tr>
<td>11</td>
<td>Representational Learning</td>
</tr>
<tr>
<td>12</td>
<td>Deep Generative Models</td>
</tr>
<tr>
<td>13</td>
<td>Deep Reinforcement Learning</td>
</tr>
<tr>
<td>14</td>
<td>Deep Learning and the Brain</td>
</tr>
<tr>
<td>15</td>
<td>Conclusions and Future Perspective</td>
</tr>
<tr>
<td>16</td>
<td>Project presentations</td>
</tr>
</tbody>
</table>

**Foundations**

**Basics**

**Advanced**

**Selected topics**