INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

Neural Networks: Regularization and Optimization

Dr. Partha Pratim Roy

Dept. of Computer Science and Engineering
Indian Institute of Technology, Roorkee, India

Associate Editor of SN Computer Science

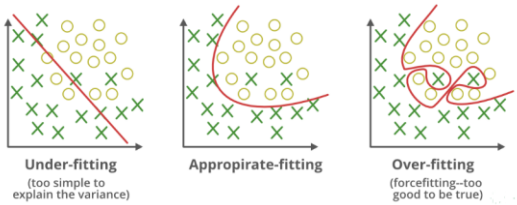
http://parimal.iitr.ac.in email: proy.fcs@iitr.ac.in

Model (Function) Fitting

- How well a model performs on training/evaluation datasets will define its characteristics

	Underfit	Overfit	Good Fit
Training Dataset	Poor	Very Good	Good
Evaluation Dataset	Very Poor	Poor	Good

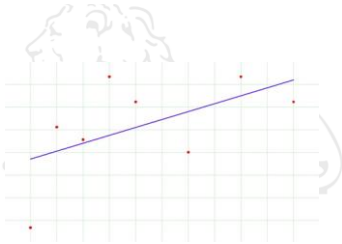
Model Fitting – Visualization



Variation in model fitting

Overfitting

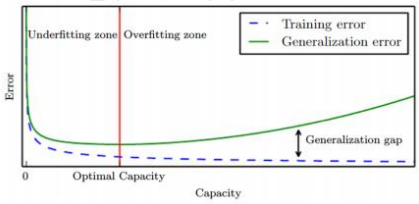
The overfitting happens when model learns signal as well as noise in the training data. It causes its performance on new data on which model wasn't trained on.



Regularization

- *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

- avoid overfitting of model.



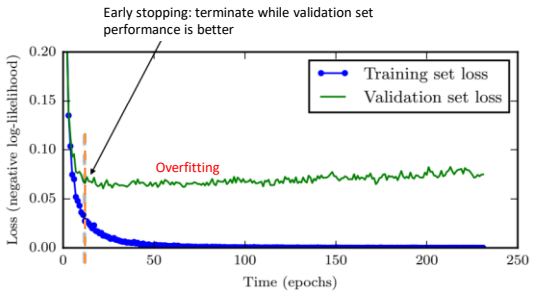
Regularization Techniques

- ❑ **Early Stopping**
- ❑ **Norm Penalties**
- ❑ **Dropout**
- ❑ Batch Norm
- ❑ **Data Augmentation**
- ❑ Layer Norm
- ❑ Weight Norm

Early Stopping

- ❑ There is point during training a large neural net when the model will stop generalizing and only focus on learning the statistical noise in the training dataset.
- ❑ Solution
  Stop whenever validation errors increases

Early Stopping



Norm Penalties

It adds the penalty as model complexity increases.

Regularization parameter (lambda) penalizes all the parameters except intercept so that model generalizes the data and won't overfit.

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$
$$\min_\theta J(\theta)$$

In above equation as the complexity is increasing, regularization will add the penalty for higher terms. This will decrease the importance given to higher terms and will bring the model towards less complex equation.

Norm Penalties

L2 adds "***squared magnitude***" of coefficient as penalty term to the loss function.

$$Loss = Loss + \lambda\sum\beta^2 \qquad \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p}x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

L1 adds "***absolute value of magnitude***" of coefficient as penalty term to the loss function.

$$Loss = Loss + \lambda\sum|\beta| \qquad \sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p}X_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|$$

Weight Penalties → Smaller Weights → Simpler Model → Less Overfit

If *lambda* is zero then we will get back OLS whereas very large value will make coefficients zero hence it will under-fit.

Norm Penalties

L1 regularization technique is called ***Lasso Regression***

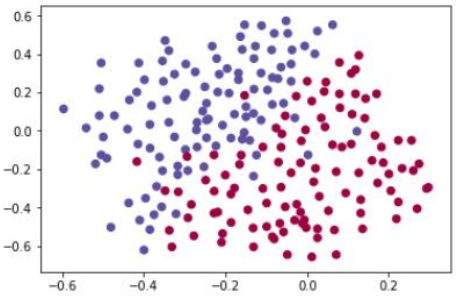
(Least Absolute Shrinkage and Selection Operator)

L2 regularization technique is called ***Ridge Regression***.

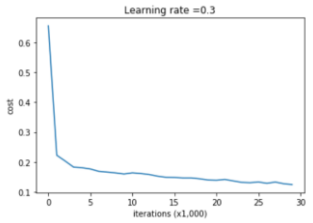
The **key difference** between these techniques is that Lasso shrinks the less important feature's coefficient to zero thus, removing some feature altogether.

- Thus, it works well for **feature selection** in case we have a huge number of features.

Norm Penalties



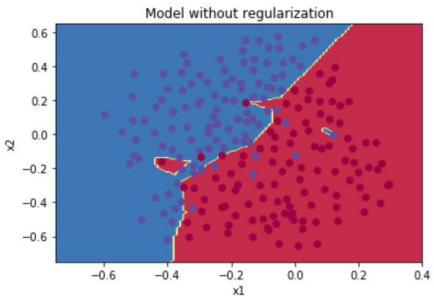
Norm Penalties




```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

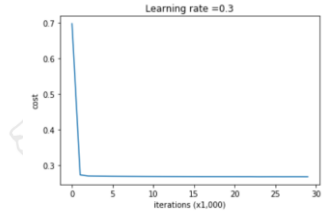
Baseline performance of a non-regularized model

Norm Penalties



Decision Boundary without regularization

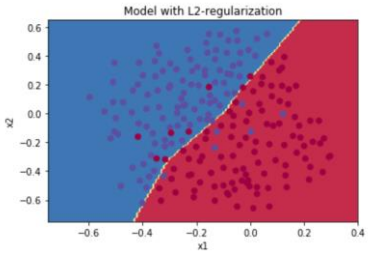
Norm Penalties




```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

Baseline performance with L2 regularization, $\lambda = 0.7$

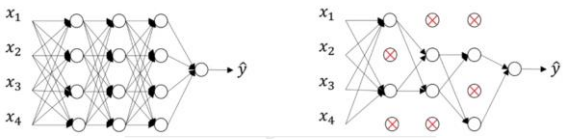
Norm Penalties



Decision Boundary with L2 regularization

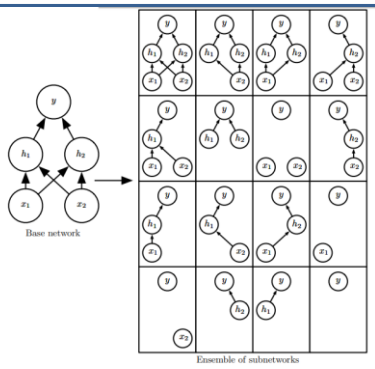
Dropout

- Removing units from base model effectively creates a subnetwork.
- All those subnetworks are trained implicitly together with all parameters shared (different from bagging)
- At predict mode, all learned units are activated, which averages all trained subnetworks

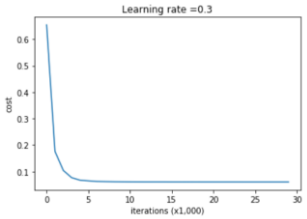


Left: neural network before dropout. Right: neural network after dropout.

Dropout



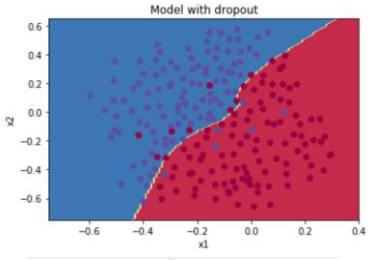
Dropout




```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```

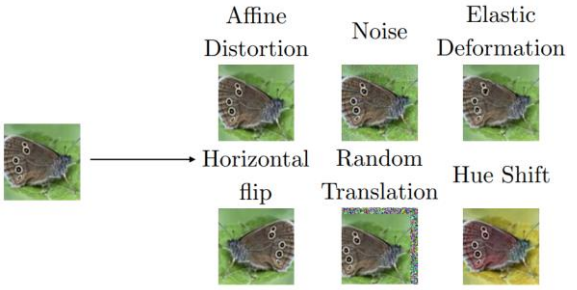
Baseline performance of Dropout with threshold of 0.8

Dropout



Decision boundary with dropout

Dataset Augmentation



OPTIMIZATION

Gradient Descent vs. SGD

- Gradient descent: **all examples at once**

$$x_{t+1} = x_t - \alpha_t \frac{1}{N} \sum_{i=1}^{N} \nabla f(x_t; y_i)$$

- Stochastic gradient descent: **one example at a time**

$$x_{t+1} = x_t - \alpha_t \nabla f(x_t; y_{i_t})$$

Mini-Batch Stochastic Gradient Descent

- An intermediate approach

$$x_{t+1} = x_t - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f(x_t; y_i)$$

where B_t is sampled uniformly from the set of all subsets of $\{1, \ldots, N\}$ of size b.

- The b parameter is the **batch size**
- Typically choose b << N.

- Also called **mini-batch gradient descent**

Advantages of Mini-Batch

- Takes **less time to compute each update** than gradient descent
  - Only needs to sum up b gradients, rather than N

$$x_{t+1} = x_t - \alpha_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f(x_t; y_i)$$

- But takes **more time for each update** than SGD

SGD Algorithm

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

The learning rate ϵ is an essential parameter. In practice, it is necessary to gradually decrease the learning rate over time.

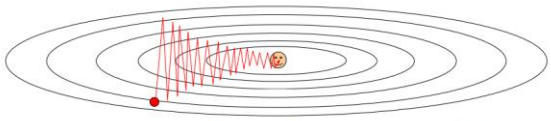
The *sufficient* conditions for guaranteed convergence of SGD is that: $\sum_{k=1}^{\infty} \epsilon_k = \infty$ $\sum_{k=1}^{\infty} \epsilon_k^2 < 0$

Gradient Descent Optimization Algorithms

- ❑ Momentum
- ❑ Nesterov accelerated gradient
- ❑ Adagrad
- ❑ Adadelta
- ❑ RMSprop
- ❑ Adam

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Zero gradient, gradient descent gets stuck

Momentum

SGD has trouble navigating **ravines**.
Momentum [Qian, 1999] helps SGD **accelerate**.
Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t. Momentum term γ is usually set to 0.9.

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

(Left) Vanilla SGD, (right) SGD with momentum.

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

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

Momentum

- Momentum tries to remedy the slowness of SGD especially in face of high curvature, small but consistent gradients, or noisy gradients. (Ill conditioning !)
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
- Analogous to rolling a ball with mass and gravity on the topology of the objective function.
- $\alpha \in [0, 1]$ is a hyperparameter that determines how quickly the contributions of previous gradients exponentially decay. In practice, α is set to be 0.5, 0.9 and 0.99.

Momentum

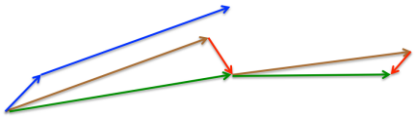
- ❑ Reduces updates for dimensions whose gradients change directions.
- ❑ Increases updates for dimensions whose gradients point in the same directions.

Optimization with momentum

Nesterov accelerated gradient

Momentum **blindly accelerates** down slopes: First computes gradient, then makes a big jump.
Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient $\theta - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

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



Nesterov update

Adagrad

- Previous methods: **Same learning rate** for all parameters
- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).
- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.
- Adagrad update:

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

$G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t
ϵ: smoothing term to avoid division by zero
$\odot$: element-wise multiplication

Adagrad

- Pros
  - Well-suited for dealing with **sparse data**.
  - Significantly **improves robustness** of SGD.
  - Lesser need to manually tune learning rate.
- Cons
  - **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

Adadelta

- Adadelta [Zeiler, 2012] restricts the window of accumulated past gradients to a **fixed size**. SGD update:

$$\Delta\theta_t = -\eta \cdot g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Defines **running average** of squared gradients $E[g^2]_t$ at time t:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

- Adagrad update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Denominator is just root mean squared (RMS) error of gradient:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

- Note: **Hypothetical units do not match.**
- Define **running average of squared parameter updates** and RMS:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Approximate with $RMS[\Delta\theta]_{t-1}$, replace η for **final Adadelta update**:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

- Developed independently from Adadelta around the same time by Geo Hinton.
- Also divides learning rate by a **running average of squared gradients.**
- RMSprop update:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

γ: decay parameter; typically set to 0.9
η: learning rate; a good default value is 0.001

Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients v_t** like Adadelta and RMSprop.
- Like Momentum, stores **running average of past gradients** m_t .

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

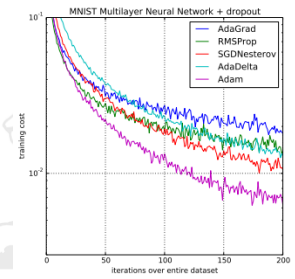
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

m_t: first moment (mean) of gradients
v_t: second moment (uncentered variance) of gradients
β_1, β_2: decay rates

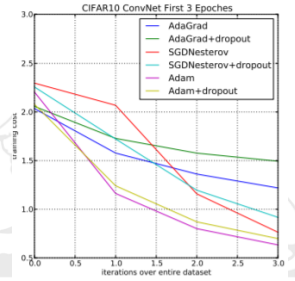
Update Equations

Method	Update equation
	$g_t = \nabla_{\theta_t} J(\theta_t)$
SGD	$\Delta\theta_t = -\eta \cdot g_t$
	$\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

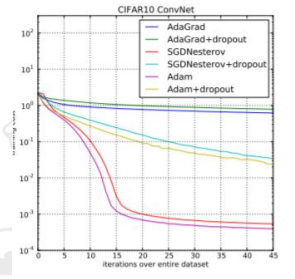
Comparison Graph



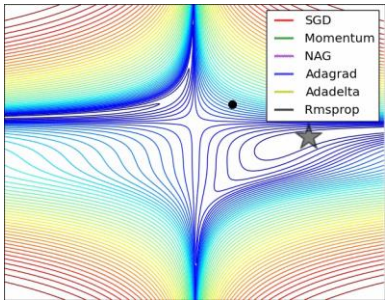
Comparison Graph



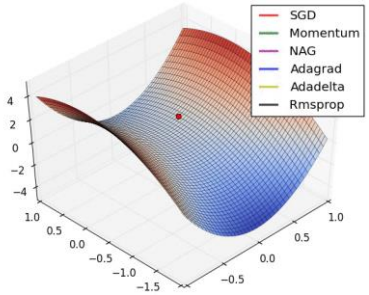
Comparison Graph



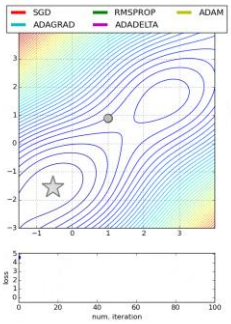
Visualization



Visualization



Visualization



Which optimizer to choose?

- Adaptive learning rate methods (Adagrad, Adadelta, RMSprop, Adam) are **particularly useful for sparse features.**
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [Kingma and Ba, 2015] show that bias-correction helps Adam **slightly outperform RMSprop.**