Loss functions

• **Many correct** labels case:
  – Binary prediction for each label, independently:
    • \( L_i = \sum_j \max(0, 1 - y_{ij}f_j) \)
    • \( y_{ij} = +1 \) if example \( i \) is labeled with label \( j \); otherwise \( y_{ij} = -1 \).

  – Alternatively, train logistic loss for each label (0 or 1):
    \[
    L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))
    \]
Previously on CENG501!
Deep learning functions

- $Wx$ - convex
  - ReLU – convex
  - Softmax – convex
  - Sigmoid, tanh – non-convex
  - Loss functions (cross-entropy, max-margin, squared-error loss) are convex
- How about NNs?
  - NNs without non-linearities are convex. The parameters just model a hyper-plane.
  - NNs with non-linearities are convex.
Large vs. small batch sizes

forth the following as possible causes for this phenomenon: (i) LB methods over-fit the model; (ii) LB methods are attracted to saddle points; (iii) LB methods lack the explorative properties of SB methods and tend to zoom-in on the minimizer closest to the initial point; (iv) SB and LB methods converge to qualitatively different minimizers with differing generalization properties. The data presented in this paper supports the last two conjectures.

The main observation of this paper is as follows:

The lack of generalization ability is due to the fact that large-batch methods tend to converge to \textit{sharp minimizers} of the training function. These minimizers are characterized by a significant number of large positive eigenvalues in $\nabla^2 f(x)$, and tend to generalize less well. In contrast, small-batch methods converge to \textit{flat minimizers} characterized by having numerous small eigenvalues of $\nabla^2 f(x)$. We have observed that the loss function landscape of deep neural networks is such that large-batch methods are attracted to regions with sharp minimizers and that, unlike small-batch methods, are unable to escape basins of attraction of these minimizers.

![Training Function](image)

\textbf{On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima}

\begin{itemize}
  \item Nitisht Shrirish Keskar
    Northwestern University
    Evanston, IL 60208
    keskar.nitisht@northwestern.edu
  \item Dhavatsa Mudigere
    Intel Corporation
    Bangalore, India
    dhavatsa.mudigere@intel.com
  \item Jorge Nocedal
    Northwestern University
    Evanston, IL 60208
    j-nocedal@northwestern.edu
  \item Mikhail Simonyan
    Intel Corporation
    Santa Clara, CA 95054
    mikhail.simonyan@intel.com
  \item Ping Tak Peter Tang
    Intel Corporation
    Santa Clara, CA 95054
    peter.tang@intel.com
\end{itemize}

ICLR, 2017
Large vs. small batch sizes

Stability [1]:

– Large batch sizes introduce stability in terms of gradient directions. But this increases the changes of getting stuck in local minima.
– Small batch sizes introduce noisy gradients which make it difficult to get stuck in local minima.

• Local convergence & width of the minima [2]:

– Small batch sizes tend to converge to solutions that are farther away from the initial position whereas large batch sizes lead to solutions close to the initial position.


Newton’s method for optimization

• $w \leftarrow w - [Hf(w)]^{-1} \nabla f(w)$
  – Makes bigger steps in shallow curvature
  – Smaller steps in steep curvature

• Note that there is no (learning rate) hyper-parameter! (if you wish you can add a step size, but this is not necessary)

• Disadvantage:
  – Too much memory requirement
  – For 1 million parameters, this means a matrix of 1 million x 1 million $\Rightarrow \sim 3725$ GB RAM
  – Alternatives exist to get around the memory problem (quasi-Newton methods, Limited-memory BFGS -- short for Broyden–Fletcher–Goldfarb–Shanno)
RPROP (Resilience Propagation)

- Instead of the magnitude, use the sign of the gradients

\[
\Delta_{ij}^{(t)} = \begin{cases} 
\eta^+ \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\
\eta^- \cdot \Delta_{ij}^{(t-1)}, & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\
\Delta_{ij}^{(t-1)}, & \text{else}
\end{cases}
\]

where \(0 < \eta^- < 1 < \eta^+\)

- Motivation: If the sign of a gradient has changed, that means we have “overshot” a minima
- Advantage: Faster to run/converge
- Disadvantage: More complex to implement
Previously on CENG501!

Figure 8: Here, the method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$.  

Jonathan Richard Shewchuk
Conjugate Gradient Descent

• Motivation
Steepest and Conjugate Gradient Descent: Cons and Pros

- **Pros:**
  - Faster to converge than, e.g., stochastic gradient descent (even mini-batch)
- **Cons:**
  - They don’t work well on saddle points
  - Computationally more expensive

Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).
Momentum

• Maintain a “memory”

$$\Delta w(t + 1) \leftarrow \mu \Delta w(t) - \eta \nabla L$$

where $\mu$ is called the momentum weight

• Momentum filters oscillations on gradients (i.e., oscillatory movements on the error surface)

• $\mu$ is typically initialized to 0.9.
  – It is better if it anneals from 0.5 to 0.99 over multiple epochs
Momentum

Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.
Nesterov Momentum

- Use a “lookahead” step to update:

\[
\begin{align*}
\mathbf{w}_{\text{ahead}} & \leftarrow \mathbf{w} + \mu \Delta \mathbf{w}(t) \\
\Delta \mathbf{w}(t + 1) & \leftarrow \mu \Delta \mathbf{w}(t) - \eta \nabla L_{\text{ahead}} \\
\mathbf{w} & \leftarrow \mathbf{w} + \Delta \mathbf{w}(t + 1)
\end{align*}
\]


Previously on CENG501!
Momentum vs. Nesterov Momentum

• When the learning rate is very small, they are equivalent.
• When the learning rate is sufficiently large, Nesterov Momentum performs better (it is more responsive).
• See for an in-depth comparison:

On the importance of initialization and momentum in deep learning

Ilya Sutskever¹
James Martens
George Dahl
Geoffrey Hinton

¹ILYASU@GOOGLE.COM
J.MARTENS@CS.TORONTO.EDU
GDAHL@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
Setting the learning rate

- Single global learning rate
  - Adaptive Learning Rate
  - Adaptive Learning Rate with Momentum

- Per-parameter learning rate
  - AdaGrad
  - RMSprop
  - Adam
  - AdaDelta
Global Methods: Scheduling the learning rate

• Step decay
  – \( \eta' \leftarrow \eta \times c \), where \( c \) could be 0.5, 0.4, 0.3, 0.2, 0.1 etc.

• Exponential decay:
  – \( \eta = \eta_0 e^{-kt} \), where \( t \) is iteration number
  – \( \eta_0, k \): hyperparameters

• 1/t decay:
  – \( \eta = \frac{\eta_0}{1 + kt} \)

• If you have time, keep decay small and train longer
Global Methods: Scheduling the learning rate

Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

Global Methods: learning rate & the batch size

- Bigger batch size, bigger learning rate
  - If you increase batch size to kN, learning rate should be scaled by:
    - $\sqrt{k}$ [1]
    - $k$ [2]
- Two interpretations:
  - Bigger batch means more stable gradient => Safer to make large steps.
  - Bigger batch means less number of update steps => increase learning rate to compensate.


Global Methods: learning rate & the batch size

BATCH SIZE:

[Graph showing the relationship between learning rate and batch size for different batch sizes (BS = 64, BS = 32, BS = 16, BS = 8, BS = 4).]

Global Methods: warm-up

• Start with a small learning rate [1]
  – Constant learning rate
  – Gradually increasing

• Why? The first steps of learning appear to be very critical [2]


### Per-parameter Methods: Adagrad

- Higher the gradient, lower the learning rate
- Accumulate square of gradients elementwise (initially $r = 0$):
  \[
  r \leftarrow r + \left( \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2
  \]
- Update each parameter/weight based on the gradient on that:
  \[
  \Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1:M} \frac{\partial L(x_i; W, b)}{\partial W}
  \]

---

**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

initial $\delta + \sqrt{r}$

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

1. Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$.
2. Accumulate squared gradient: $r \leftarrow r + g \odot g$.
3. Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$. (Division and square root applied element-wise)
4. Apply update: $\theta \leftarrow \theta + \Delta \theta$.

---


Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:

Root-Mean-Squared Propagation (RMSprop)

- Similar to Adagrad. Adagrad uses the whole history of gradients, which can be a limitation when training converges to a nice “bowl”.

- RMSprop handles this by weighted averaging:
  \[ r \leftarrow \rho r + (1 - \rho) \left( \sum_{i=1}^{M} \frac{\partial L(x_i; W, b)}{\partial W} \right)^2 \]
  \[ r' = r \]

- \( \rho \) is typically [0.9, 0.99, 0.999]

- Update each parameter/weight based on the gradient on that:
  \[ \Delta W \leftarrow -\frac{\eta}{\sqrt{r}} \sum_{i=1}^{M} \frac{\partial L(x_i; W, b)}{\partial W} \]

\[ \eta \]

**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: \( \Delta \theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g \) (applied element-wise)

Apply update: \( \theta \leftarrow \theta + \Delta \theta \).

end while

Currently, unpublished. Proposed by Hinton in one of his lectures.

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:
RMSprop with Nesterov Momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate $\epsilon$, decay rate $\rho$, momentum coefficient $\alpha$

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

Initialize 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 interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$.

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

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

Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$. ($\frac{1}{\sqrt{r}}$ applied element-wise)

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

end while

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.
Per-parameter Methods:
Adaptive Moments (Adam)

- A variation of RMSprop + momentum
- Incorporates first & second order moments
- Bias correction needed to get rid of bias towards zero at initialization

Algorithm 8.7 The Adam algorithm

Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default: $10^{-8}$)

Require: Initial parameters $\theta$

Initialize 1st and 2nd moment variables $s = 0, r = 0$
Initialize time step $t = 0$

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: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)g$
Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$
Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$
Compute update: $\Delta \theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r} + \delta}}$ (operations applied element-wise)
Apply update: $\theta \leftarrow \theta + \Delta \theta$

end while
Today

• Towards deep learning
  – Representational Capacity
  – Overfitting, convergence, when to stop training
  – Preprocessing and initialization
  – Issues and practical advices

• Convolutional Neural Networks
  – CNNs vs. MLPs
  – Operations in CNNs
  – Convolution in CNNs

Administrative Issues

• Programming assignment 1: Deadline 29\textsuperscript{th} of April
• Take-Home Exam 1:
  – Announced on Friday, 30\textsuperscript{th} of April
  – Deadline: 2\textsuperscript{nd} of May?
• Project paper selection
  – [https://docs.google.com/spreadsheets/d/1tzPHq_Vgu6gCwNyXJHGvqeA6pgU67H0nKYjqkisWfKc/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1tzPHq_Vgu6gCwNyXJHGvqeA6pgU67H0nKYjqkisWfKc/edit?usp=sharing)
    – Deadline: 19\textsuperscript{th} of April
REPRESENTATIONAL CAPACITY
Representational capacity

• Boolean functions:
  – Every Boolean function can be represented exactly by a neural network
  – The number of hidden layers might need to grow with the number of inputs
• Continuous functions:
  – Every bounded continuous function can be approximated with small error with two layers
• Arbitrary functions:
  – Three layers can approximate any arbitrary function


Representational Capacity: Why go deeper if 3 layers is sufficient?

• Going deeper helps convergence in “big” problems.
• Going deeper in “old-fashion trained” ANNs does not help much in accuracy
  – However, with different training strategies or with Convolutional Networks, going deeper matters

Representational Capacity

• More hidden neurons $\Rightarrow$ capacity to represent more complex functions

• Problem: overfitting vs. generalization
  – We will discuss the different strategies to help here (L2 regularization, dropout, input noise, using a validation set etc.)

Figure: https://cs231n.github.io/
Number of hidden neurons

Several rule of thumbs (Jeff Heaton)

– The number of hidden neurons should be between the size of the input layer and the size of the output layer.

– The number of hidden neurons should be $2/3$ the size of the input layer, plus the size of the output layer.

– The number of hidden neurons should be less than twice the size of the input layer.
Number of hidden layers

• Depends on the nature of the problem
  – Linear classification? ➔ No hidden layers needed
  – Non-linear classification?
Model Complexity

• Models range in their flexibility to fit arbitrary data

- High bias
  - Large capacity may allow it to memorize data and fail to capture regularities
  - Small capacity may prevent it from representing all structure in data

- Low bias
  - High variance
  - Low variance
Training Vs. Test Set Error

Error vs. Model Complexity

Optimum Model Complexity

Training Set

Test Set

Slide Credit: Michael Mozer
Bias-Variance Trade Off

![Graph showing the trade off between bias and variance in model complexity. The graph illustrates how error on the test set changes as model complexity varies. The minimum error occurs at the optimum model complexity.](image credit: scott.fortmann-roe.com)

Underfit: Model complexity is too low, leading to high bias.

Overfit: Model complexity is too high, leading to high variance.

Slide Credit: Michael Mozer
Double Descent

Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

Double Descent

What do the layers represent?

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]
What do the layers represent?

**FIGURE 4.7**
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

OVERFITTING, CONVERGENCE, AND WHEN TO STOP
Overfitting

• Occurs when training procedure fits to not only regularities in training data but also noise.
  – Like memorizing the training examples instead of learning the statistical regularities
• Leads to poor performance on test set
• Most of the practical issues with neural nets involve avoiding overfitting

Figure: https://cs231n.github.io/

Adapted from Michael Mozer
Avoiding Overfitting

• Increase training set size
  – Make sure effective size is growing; redundancy doesn’t help

• Incorporate domain-appropriate bias into model
  – Customize model to your problem

• Set hyperparameters of model
  – number of layers, number of hidden units per layer, connectivity, etc.

• Regularization techniques
Incorporating Domain-Appropriate Bias Into Model

• Input representation
• Output representation
• Architecture
  – # layers, connectivity
  – e.g., convolutional nets, residual connections etc.
• Activation function
• Loss function
Customizing Networks

• Neural nets can be customized based on the problem domain
  – choice of loss function
  – choice of activation function

• Domain knowledge can be used to impose domain-appropriate bias on model
  – bias is good if it reflects properties of the data set
  – bias is harmful if it conflicts with properties of data
Adding bias into a model

• Adding hidden layers or direct connections based on the problem

Slide Credit: Michael Mozer
Adding bias into a model

• Modular architectures
  – Specialized hidden units for special problems
Adding bias into a model

• Local or specialized receptive fields
  – E.g., in CNNs
• Constraints on activities
• Constraints on weights

Slide Credit: Michael Mozer
Adding bias into a model

• Use different loss functions (e.g., cross-entropy)
• Use specialized activation functions
Adding bias into a model

• Introduce other parameters
  – Temperature
  – Saliency of input
Regularization

- Regularization strength can affect overfitting

\[ \frac{1}{2} \lambda w^2 \]

Figure: https://cs231n.github.io/
Regularization

• L2 regularization: \( \frac{1}{2} \lambda w^2 \)
  – Very common
  – Penalizes peaky weight vector, prefers diffuse weight vectors

• L1 regularization: \( \lambda |w| \)
  – Enforces sparsity (some weights become zero)
  – Why? Weight decay is by a constant value if \(|w|\) is non-zero.
  – Leads to input selection (makes it noise robust)
  – Use it if you require sparsity / feature selection

• Can be combined: \( \lambda_1 |w| + \lambda_2 w^2 \)

• Regularization is not performed on the bias; it seems to make no significant difference
L2 regularization and weight decay

• L2 regularization

\[ w_i \leftarrow w_i - \eta \left( \frac{\partial L_{\text{data}}}{\partial w_i} + \lambda w_i \right) \]

• Weight decay

\[ w_i \leftarrow w_i - \eta \frac{\partial L_{\text{data}}}{\partial w_i} - \alpha w_i \]
Weight Decay

• Adam & weight decay issue:

https://www.fast.ai/2018/07/02/adam-weight-decay/
L0 regularization

- \( L_0 = \left( \sum_i x_i^0 \right)^{1/0} \)
- How to compute the zeroth power and zeroth-root?
- Mathematicians approximate this as:
  - \( L_0 = \# \{ i \mid x_i \neq 0 \} \)
  - The cardinality of non-zero elements
- This is a strong enforcement of sparsity.
- However, this is non-convex
  - L1 norm is the closest convex form
Regularization

• Enforce an upper bound on weights:
  – Max norm:
    • $\|w\|_2 < c$
    • Helps the gradient explosion problem
    • Improvements reported

• Dropout:
  – At each iteration, drop a number of neurons in the network
  – **Use a neuron’s activation** with probability $p$ (a hyperparameter)
  – Adds stochasticity!

Fig: Srivastava et al., 2014

Regularization: Dropout

- Feed-forward only on active units
- Can be trained using SGD with mini-batch
  - Back propagate only “active” units.
- One issue:
  - Expected output $x$ with dropout:
    - $E[x] = px + (1 - p)0$
- To have the same scale at testing time (no dropout), multiply test-time activations with $p$.

Fig: Srivastava et al., 2014
Regularization: Dropout

Training-time:

```python
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p  # first dropout mask
H1 *= U1  # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p  # second dropout mask
H2 *= U2  # drop!
out = np.dot(W3, H2) + b3
```

Test-time: All neurons receive their normal input \(x\) so we should scale by \(p\) to have \(E[x] = px\).

```python
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
out = np.dot(W3, H2) + b3
```
Regularization: Inverted Dropout

Perform scaling while dropping at training time!

**Training-time:** Correct the expected output from $px$ to $x$.

```python
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p  # first dropout mask. Notice /p!
H1 *= U1  # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p  # second dropout mask. Notice /p!
H2 *= U2  # drop!
out = np.dot(W3, H2) + b3
```

**Test-time:**

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)  # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

Figure 1: The $\ell_2$-regularized cross-entropy train loss surface of a ResNet-164 on CIFAR-100, as a function of network weights in a two-dimensional subspace. In each panel, the horizontal axis is fixed and is attached to the optima of two independently trained networks. The vertical axis changes between panels as we change planes (defined in the main text). **Left:** Three optima for independently trained networks. **Middle** and **Right:** A quadratic Bezier curve, and a polygonal chain with one bend, connecting the lower two optima on the left panel along a path of near-constant loss. Notice that in each panel a direct linear path between each mode would incur high loss.


Drop-Activation: Implicit Parameter Reduction and Harmonic Regularization

Senwei Liang  
National University of Singapore  
10 Lower Kent Ridge Road  
Singapore 119076  
liangsenwei@u.nus.edu

Yuehaw Kwoo  
Stanford University  
450 Serra Mall  
Stanford, CA 94305  
ykhoo@stanford.edu

Haizhao Yang  
National University of Singapore  
10 Lower Kent Ridge Road  
Singapore 119076  
haizhao@nus.edu.sg

14 November 2018

(a) Standard neural network with nonlinearity

(b) After applying Drop-Activation

<table>
<thead>
<tr>
<th>model</th>
<th>Baseline</th>
<th>DropAct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-164</td>
<td>8.85</td>
<td>8.82</td>
</tr>
<tr>
<td>PreResNet-164</td>
<td>8.88</td>
<td>8.72</td>
</tr>
<tr>
<td>WideResNet-28-10</td>
<td>8.97</td>
<td>8.72</td>
</tr>
<tr>
<td>DenseNet-BC-100-12</td>
<td>8.81</td>
<td>8.90</td>
</tr>
<tr>
<td>ResNeXt-29-8x64d</td>
<td>9.07</td>
<td>8.91</td>
</tr>
</tbody>
</table>

Table 4: Test error (%) on EMNIST (Balanced). The Baseline results were generated by ourselves.
“Dropout performs gradient descent on-line with respect to both the training examples and the ensemble of all possible subnetworks.”

Dropout is a special case of the stochastic delta rule: faster and more accurate deep learning

Noah Frazier-Logue  
Rutgers University Brain Imaging Center  
Rutgers University - Newark  
Newark, NJ 07103  
n.frazier.logue@nyu.edu

Stephen José Hanson  
Rutgers University Brain Imaging Center  
Rutgers University - Newark  
Newark, NJ 07103  
jose@rubic.rutgers.edu

\[
S(w_{ij} = \tilde{w}_{ij}) = \mu_{w_{ij}} + \mu_{w_{ij}} \theta(w_{ij}; 0, 1)
\]

The first update rule refers to the mean of the weight distribution:

\[
\mu_{w_{ij}}(n + 1) = \alpha \left( \frac{\partial E}{\partial \tilde{w}_{ij}} \right) + \mu_{w_{ij}}(n)
\]

and is directly dependent on the error gradient and has learning rate \( \alpha \). This is the usual delta rule update but conditioned on sample weights thus causing weight sharing through the updated mean value. The second update rule is for the standard deviation of the weight distribution (and for a Gaussian is known to be sufficient for identification).

\[
\sigma_{w_{ij}}(n + 1) = \beta \left| \frac{\partial E}{\partial \tilde{w}_{ij}} \right| + \sigma_{w_{ij}}(n)
\]

Figure 1: SDR sampling.
Lottery Ticket Hypothesis


**Identifying winning tickets.** We identify a winning ticket by training a network and pruning its smallest-magnitude weights. The remaining, unpruned connections constitute the architecture of the winning ticket. Unique to our work, each unpruned connection’s value is then reset to its initialization from original network before it was trained. This forms our central experiment:

1. Randomly initialize a neural network \( f(x; \theta_0) \) (where \( \theta_0 \sim D_\theta \)).
2. Train the network for \( j \) iterations, arriving at parameters \( \theta_j \).
3. Prune \( p\% \) of the parameters in \( \theta_j \), creating a mask \( m \).
4. Reset the remaining parameters to their values in \( \theta_0 \), creating the winning ticket \( f(x; m \odot \theta_0) \).

As described, this pruning approach is one-shot: the network is trained once, \( p\% \) of weights are pruned, and the surviving weights are reset. However, in this paper, we focus on iterative pruning, which repeatedly trains, prunes, and resets the network over \( n \) rounds; each round prunes \( p^{\frac{1}{n}} \% \) of the weights that survive the previous round. Our results show that iterative pruning finds winning tickets that match the accuracy of the original network at smaller sizes than does one-shot pruning.
Data Augmentation

Regularization Summary

• L2 regularization
• Inverted dropout with $p = 0.5$ (tunable)
• Data augmentation
When To Stop Training

1. Train $n$ epochs; lower learning rate; train $m$ epochs
   - bad idea: can’t assume one-size-fits-all approach

2. Loss-change criterion
   - stop when loss isn’t dropping
   - recommendation: criterion based on % drop over a window of, say, 10 epochs
     - 1 epoch is too noisy
     - absolute error criterion is too problem dependent
   - Another idea: train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
When To Stop Training

• 3. Weight-change criterion
  – Compare weights at epochs \((t - 10)\) and \(t\) and test:
    \[
    \max_i |w_i^t - w_i^{t-10}| < \theta
    \]
  – Don’t base on length of overall weight change vector
  – Possibly express as a percentage of the weight
  – Be cautious: small weight changes at critical points can result in rapid drop in error
Training Vs. Test Set Error

Trainig Set

Test Set
DATA PREPROCESSING AND WEIGHT INITIALIZATION
Data Preprocessing

- Mean subtraction
- Normalization
- PCA and whitening
Data Preprocessing: Mean subtraction

• Compute the mean of each dimension, $\mu_i$, over the training set:
  \[ \mu_i = \frac{1}{N} \sum_j x_{ji} \]

• Subtract the mean for each dimension:
  \[ x'_{ji} \leftarrow x_{ji} - \mu_i \]

• Effect: Move the data center (mean) to coordinate center

![Original data vs. zero-centered data](http://cs231n.github.io/neural-networks-2/)

Mean image of CIFAR10 (from PA1)
Data Preprocessing: Normalization (or conditioning)

- Necessary if you believe that your dimensions have different scales
  - Might need to reduce this to give equal importance to each dimension
- Normalize each dimension by its std. dev. after mean subtraction:
  \[ x'_{ji} = x_{ji} - \mu_i \]
  \[ x''_{ji} = x'_{ji} / \sigma_i \]
- Effect: Make the dimensions have the same scale

Data Preprocessing:
Principle Component Analysis

• First center the data
• Find the eigenvectors $e_1, ..., e_n$
• Project the data onto the eigenvectors:
  \[ x_i^R = x_i \cdot [e_1, ..., e_n] \]
• This corresponds to rotating the data to have the eigenvectors as the axes
• If you take the first $M$ eigenvectors, it corresponds to dimensionality reduction

Data Preprocessing: Whitening

• Normalize the scale with the norm of the eigenvalue:

\[ x_i^w = x_i^R / (\lambda_i + \epsilon) \]

• \( \epsilon \): a very small number to avoid division by zero
• This stretches each dimension to have the same scale.
• Side effect: this may exaggerate noise.

Data Preprocessing: Example

![Image of original images, top 144 eigenvectors, reduced images, and whitened images]

Data Preprocessing: Summary

• We mostly don’t use PCA or whitening
  – They are computationally very expensive
  – Whitening has side effects
• It is quite crucial and common to zero-center the data
• Most of the time, we see normalization with the std. deviation
Weight Initialization

• Zero weights
  – Wrong!
  – Leads to updating weights by the same amounts for every input
  – Symmetry!

• Initialize the weights randomly to small values:
  – Sample from a small range, e.g., Normal(0,0.01)
  – Don’t initialize too small

• The bias may be initialized to zero
  – For ReLU units, this may be a small number like 0.01.

Note: None of these provide guarantees. Moreover, there is no guarantee that one of these will always be better.
Initial Weight Normalization

• Problem: Variance of the output changes with the number of inputs

• If $s = \sum_i w_i x_i$ (note that $Var(X) = E[(X - \mu)^2]$):

\[
Var(s) = Var(\sum_i^nw_ix_i) = \sum_i^nVar(w_i x_i) = \sum_i^n[E(w_i)]^2Var(x_i) + E[(x_i)]^2Var(w_i) + Var(x_i)Var(w_i) = \sum_i^nVar(x_i)Var(w_i) = (nVar(w))Var(x)
\]

Eqn: https://cs231n.github.io/neural-networks-2/#init
Initial Weight Normalization

• **Solution:**
  – Get rid of $n$ in $Var(s) = (n Var(w))Var(x)$

• **How?**
  – $w_i = \text{rand}(0, 1)/\sqrt{n}$
  – Why? Because: $Var(aX) = a^2 Var(X)$

• **Glorot & Bengio** *(for symmetric activation functions)*:
  \[ w_i = \text{rand}(0, 1) \times \frac{\sqrt{2}}{\sqrt{n_{in}} + n_{out}} \]

with scale correction
*(also known as Xavier initialization)*:

\[ w_i = \text{rand}(0, 1) \times \frac{\sqrt{6}}{\sqrt{n_{in}} + n_{out}} \]

---

Figure 6: Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.

Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

Initial Weight Normalization

• He et al. shows that Xavier initialization does not work well for ReLUs.

They suggest the following:

\[ w_i = \text{rand}(0,1) \times \frac{\sqrt{2}}{\sqrt{n}} \]

Figure 2. The convergence of a 22-layer large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and "Xavier" (blue) [7] lead to convergence, but ours starts reducing error earlier.

Figure 3. The convergence of a 30-layer small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But "Xavier" (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

Alternative: Batch Normalization

- Normalization is differentiable
  - So, make it part of the model (not only at the beginning)
  - I.e., perform normalization during every step of processing
- More robust to initialization
- Shown to also regularize the network in some cases (dropping the need for dropout)
- Issue: How to normalize at test time?
  1. Store means and variances during training, or
  2. Calculate mean & variance over your test data
- PyTorch: use model.eval() in test time.

\[
\text{Input: } \text{Values of } x \text{ over a mini-batch: } \mathcal{B} = \{x_1...m\}; \\
\text{Parameters to be learned: } \gamma, \beta \\
\text{Output: } \{y_i = \text{BN}_{\gamma,\beta}(x_i)\}
\]

\[
\begin{align*}
\mu_{\mathcal{B}} & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i & \text{// mini-batch mean} \\
\sigma_{\mathcal{B}}^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 & \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} & \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) & \text{// scale and shift}
\end{align*}
\]

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

Alternative: Batch Normalization

• Before or after non-linearity?
• Proposers
  – “If, however, we could ensure that the distribution of nonlinearity inputs remains more stable as the network trains, then the optimizer would be less likely to get stuck in the saturated regime, and the training would accelerate.”
  – “As each layer observes the inputs produced by the layers below, it would be advantageous to achieve the same whitening of the inputs of each layer. By whitening the inputs to each layer, we would take a step towards achieving the fixed distributions of inputs that would remove the ill effects of the internal covariate shift.”
  – “We add the BN transform immediately before the nonlinearity, by normalizing \( x = Wu + b \). We could have also normalized the layer inputs \( u \), but since \( u \) is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, \( Wu + b \) is more likely to have a symmetric, non-sparse distribution, that is “more Gaussian” (Hyv"arinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution.”
Alternative Normalizations

https://medium.com/syncedreview/facebook-ai-proposes-group-normalization-alternative-to-batch-normalization-fb0699bffae7
Since we get a clear knowledge about the disharmony between Dropout and BN, we can easily develop several approaches to combine them together, to see whether an extra improvement could be obtained. In this section, we introduce two possible solutions in modifying Dropout. One is to avoid the scaling on feature-map before every BN layer, by only applying Dropout after the last BN block. Another is to slightly modify the formula of Dropout and make it less sensitive to variance, which can alleviate the shift problem and stabilize the numerical behaviors.
How critical are BatchNorm parameters?

Figure 1. Test accuracy when training all parameters of the deep (left) and wide (right) ResNets in Table 1 with $\gamma$ and $\beta$ enabled (blue) and frozen at their initial values (purple). Except on the deepest ResNets, accuracy is about half a percent lower when $\gamma$ and $\beta$ are disabled.

Figure 3. Test accuracy of ResNet-110 during training when training all parameters, just BatchNorm, and BatchNorm with shortcut and output parameters. Learning appears to occur at a similar rate in all experiments, although they reach different accuracies.

To sum up

- Initialization and normalization are crucial
- Different initialization & normalization strategies may be needed for different deep learning methods
  - E.g., in CNNs, normalization might be performed only on convolution etc.
• Vanishing gradient
  – Saturated units block gradient propagation (why?)
  – A problem especially present in recurrent networks or networks with a lot of layers
• Overfitting
  – Drop-out, regularization and other tricks.
• Tricks:
  – Unsupervised pretraining
• Batch normalization (each unit’s preactivation is normalized)
  – Helps keeping the preactivation non-saturated
  – Do this for mini-batches (adds stochasticity)
  – Backprop needs to be updated
Unsupervised pretraining

Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left**: 1 hidden layer. **Right**: 4 hidden layers.
Unsupervised pretraining

Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to zero, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments performed by Dumitru Erhan.
What if things are not working?

• Check your gradients by comparing them against numerical gradients
  – Check whether you are using an appropriate floating point representation
    • Be aware of floating point precision/loss problems
  – Turn off drop-out and other “extra” mechanisms during gradient check
  – This can be performed only on a few dimensions

• Regularization loss may dominate the data loss
  – First disable regularization loss & make sure data loss works
  – Then add regularization loss with a big factor
  – And check the gradient in each case
What if things are not working?

• Have a feeling of the initial loss value
  – For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is $-\ln(0.1)=2.302$
  – For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).

• Try to overfit on a tiny subset of the dataset
  – The cost should reach to zero if things are working properly
What if things are not working?

Learning rate might be too low;
Batch size might be too small
What if things are not working?
What if things are not working?

• Plot the histogram of activations per layer
  – E.g., for tanh functions, we expect to see a diverse distribution of values between [-1,1]
What if things are not working?

• Visualize your layers (the weights)
Also read the following

• 37 reasons why your neural network is not working:
  – https://medium.com/@slavivanov/4020854bd607

• “A Recipe for Training Neural Networks” by Karpathy:
What is best then?

• Which algorithm to choose?
  – No answer yet
  – See Tom Schaul (2014)
  – Adam, RMSprop seem to be slightly favorable; however, no best algorithm

• SGD, SGD+momentum, RMSprop, RMSprop+momentum, Adam are the most widely used ones
Luckily, deep networks are very powerful

Regularization is turned off in the experiments. When you turn on regularization, the networks perform worse.
Concluding remarks for the first part

• Loss functions
• Gradients of loss functions for minimizing them
  – All operations in the network should be differentiable
• Gradient descent and its variants
• Initialization, normalization, adaptive learning rate, ...
• Overall, you have learned most of the tools you will use in the rest of the course.