Lecture 7: Training Neural Networks, Part II
Administrative:

- A1 grades will be released this weekend
- Project proposal are due April 24th
- A2 is due Friday April 28th, 11:59pm
Administrative: assignments

- **Submission:**
  - submit a version of your assignment the night before the deadline to make sure you have no gradescope submission issues

- **Private test cases:**
  - Gradescope contains private tests that we use to autograde your work

- **Modifying code:**
  - Do not modify code outside of the code blocks
  - Do not leave print statements in your code
Last time: Activation Functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1+e^{-x}} \]

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]

**Leaky ReLU**
\[ \max(0.1x, x) \]

**Maxout**
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[ \begin{cases} 
    x & x \geq 0 \\
    \alpha(e^x - 1) & x < 0 
\end{cases} \]
Last time: Activation Functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1+e^{-x}} \]

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]  
*Good default choice*

**Leaky ReLU**
\[ \max(0.1x, x) \]

**Maxout**
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[
\begin{cases} 
  x & \text{if } x \geq 0 \\
  \alpha(e^x - 1) & \text{if } x < 0 
\end{cases}
\]
Last time: Weight Initialization

**Initialization too small:**
Activations go to zero, gradients also zero, No learning =(

**Initialization too big:**
Activations saturate (for tanh), Gradients zero, no learning =(

**Initialization just right:**
Nice distribution of activations at all layers, Learning proceeds nicely =)
Last time: Data Preprocessing
Last Time: Batch Normalization

Input: \( x : N \times D \)

Learnable scale and shift parameters:
\( \gamma, \beta : D \)

Learning \( \gamma = \sigma \), \( \beta = \mu \) will recover the identity function!

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}\quad \text{Per-channel mean, shape is D}
\]

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}
\]

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized x, Shape is N x D}
\]

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}
\]

[Ioffe and Szegedy, 2015]
Batch Normalization: Test-Time

Input: $x : N \times D$

\[ \mu_j = \text{(Running) average of values seen during training} \]

Per-channel mean, shape is D

\[ \sigma_j^2 = \text{(Running) average of values seen during training} \]

Per-channel var, shape is D

Learnable scale and shift parameters:

\[ \gamma, \beta : D \]

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]

Normalized $x$, Shape is $N \times D$

\[ y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \]

Output, Shape is $N \times D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
Batch Normalization

- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

[Ioffe and Szegedy, 2015]
Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

\[
x: \mathbb{N} \times D
\]

\[
\mu, \sigma: 1 \times D
\]

\[
\gamma, \beta: 1 \times D
\]

\[
y = \gamma (x - \mu) / \sigma + \beta
\]

Batch Normalization for **convolutional** networks

(Spatial Batchnorm, BatchNorm2D)

\[
x: \mathbb{N} \times C \times H \times W
\]

\[
\mu, \sigma: 1 \times C \times 1 \times 1
\]

\[
\gamma, \beta: 1 \times C \times 1 \times 1
\]

\[
y = \gamma (x - \mu) / \sigma + \beta
\]
Layer Normalization

Batch Normalization for fully-connected networks

$\mathbf{x}: N \times D$

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma (x - \mu) / \sigma + \beta$

Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$\mathbf{x}: N \times D$

$\mu, \sigma: N \times 1$

$\gamma, \beta: 1 \times D$

$y = \gamma (x - \mu) / \sigma + \beta$

**Instance Normalization**

**Batch Normalization** for convolutional networks

\[
\mathbf{x} : \ N \times C \times H \times W \\
\mu, \sigma : 1 \times C \times 1 \times 1 \\
\gamma, \beta : 1 \times C \times 1 \times 1 \\
y = \gamma (x - \mu) / \sigma + \beta
\]

**Instance Normalization** for convolutional networks

\[
\mathbf{x} : \ N \times C \times H \times W \\
\mu, \sigma : N \times C \times 1 \times 1 \\
\gamma, \beta : 1 \times C \times 1 \times 1 \\
y = \gamma (x - \mu) / \sigma + \beta
\]

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017
Comparison of Normalization Layers

Wu and He, “Group Normalization”, ECCV 2018
Group Normalization

Wu and He, “Group Normalization”, ECCV 2018
Today

- Improve your training error:
  - (Fancier) Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters
(Fancier) Optimizers
Optimization

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large
Optimization: Problem #1 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
What if the loss function has a local minima or saddle point?
What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
Optimization: Problem #2 with SGD

What if the loss function has a local minima or saddle point?

Saddle points much more common in high dimension

Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014
Optimization: Problem #3 with SGD

Our gradients come from minibatches so they can be noisy!

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) \]
SGD + Momentum

Local Minima  Saddle points

Poor Conditioning

Gradient Noise

SGD  SGD+Momentum
SGD: the simple two line update code

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
SGD + Momentum:
continue moving in the general direction as the previous iterations

SGD

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

SGD+Momentum

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

while True:
    dx = compute_gradient(x)
    x = learning_rate * dx

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
**SGD + Momentum:**
continue moving in the general direction as the previous iterations

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

**SGD**

**SGD+Momentum**

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum: alternative equivalent formulation

\[
\begin{align*}
v_{t+1} &= \rho v_t - \alpha \nabla f(x_t) \\
x_{t+1} &= x_t + v_{t+1}
\end{align*}
\]

\[
\begin{align*}
v_{t+1} &= \rho v_t + \nabla f(x_t) \\
x_{t+1} &= x_t - \alpha v_{t+1}
\end{align*}
\]

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of \(x\)

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD+Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]

\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ \nu_{t+1} = \rho \nu_t - \alpha \nabla f(x_t + \rho \nu_t) \]
\[ x_{t+1} = x_t + \nu_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)

Change of variables \( \tilde{x}_t = x_t + \rho \nu_t \) and rearrange:

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \)

Change of variables \( \tilde{x}_t = x_t + \rho v_t \) and rearrange:

\[ v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t) \]
\[ \tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \]
\[ = \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \]

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

- SGD
- SGD+Momentum
- Nesterov
AdaGrad

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“Per-parameter learning rates”
or “adaptive learning rates”

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011
Q: What happens with AdaGrad?

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```
AdaGrad

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q: What happens with AdaGrad? Progress along “steep” directions is damped; progress along “flat” directions is accelerated.
Q2: What happens to the step size over long time?
Q2: What happens to the step size over long time? Decays to zero
RMSProp: “Leaky AdaGrad”

AdaGrad

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012
RMSProp

- SGD
- SGD+Momentum
- RMSProp
- AdaGrad
  (stuck due to decaying lr)
Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Adam (full form)

```python
firstMoment = 0
secondMoment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    firstMoment = beta1 * firstMoment + (1 - beta1) * dx
    secondMoment = beta2 * secondMoment + (1 - beta2) * dx * dx
    first_unbias = firstMoment / (1 - beta1 ** t)
    second_unbias = secondMoment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Bias correction for the fact that first and second moment estimates start at zero

Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\text{learning}_\text{rate} = 1e-3$ or $5e-4$ is a great starting point for many models!

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
L2 Regularization vs Weight Decay

**Optimization Algorithm**

$L(w) = L_{data}(w) + L_{reg}(w)$

$g_t = \nabla L(w_t)$

$s_t = \text{optimizer}(g_t)$

$w_{t+1} = w_t - \alpha s_t$

**L2 Regularization**

$L(w) = L_{data}(w) + \lambda |w|^2$

$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$

$s_t = \text{optimizer}(g_t)$

$w_{t+1} = w_t - \alpha s_t$
L2 Regularization vs Weight Decay

Optimization Algorithm
\[ L(w) = L_{data}(w) + L_{reg}(w) \]
\[ g_t = \nabla L(w_t) \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

L2 Regularization
\[ L(w) = L_{data}(w) + \lambda |w|^2 \]
\[ g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t \]
\[ s_t = \text{optimizer}(g_t) \]
\[ w_{t+1} = w_t - \alpha s_t \]

Weight Decay
\[ L(w) = L_{data}(w) \]
\[ g_t = \nabla L_{data}(w_t) \]
\[ s_t = \text{optimizer}(g_t) + 2\lambda w_t \]
\[ w_{t+1} = w_t - \alpha s_t \]

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!
L2 Regularization vs Weight Decay

**Optimization Algorithm**

\[
L(w) = L_{data}(w) + L_{reg}(w) \\
g_t = \nabla L(w_t) \\
s_t = \text{optimizer}(g_t) \\
w_{t+1} = w_t - \alpha s_t
\]

**L2 Regularization**

\[
L(w) = L_{data}(w) + \lambda |w|^2 \\
g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t \\
s_t = \text{optimizer}(g_t) \\
w_{t+1} = w_t - \alpha s_t
\]

**Weight Decay**

\[
L(w) = L_{data}(w) \\
g_t = \nabla L_{data}(w_t) \\
s_t = \text{optimizer}(g_t) + 2\lambda w_t \\
w_{t+1} = w_t - \alpha s_t
\]

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)
AdamW: Decoupled Weight Decay

**Algorithm 2**  Adam with L₂ regularization and Adam with decoupled weight decay (AdamW)

1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: initialize time step $t \leftarrow 0$, parameter vector $\theta_{t=0} \in \mathbb{R}^n$, first moment vector $m_{t=0} \leftarrow \theta$, second moment vector $v_{t=0} \leftarrow \theta$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: repeat
4: $t \leftarrow t + 1$
5: $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$  \hspace{1cm} $\triangleright$ select batch and return the corresponding gradient
6: $g_t \leftarrow \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$  \hspace{1cm} $\triangleright$ here and below all operations are element-wise
7: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$  \hspace{1cm} $\triangleright$ $\beta_1$ is taken to the power of $t$
8: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$  \hspace{1cm} $\triangleright$ $\beta_2$ is taken to the power of $t$
9: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
10: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$  \hspace{1cm} $\triangleright$ can be fixed, decay, or also be used for warm restarts
12: $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1} \right)$
13: until stopping criterion is met
14: return optimized parameters $\theta_t$
First-Order Optimization

Loss vs. $w_1$
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Second-Order Optimization

(1) Use gradient and Hessian to form quadratic approximation
(2) Step to the minima of the approximation
Second-Order Optimization

second-order Taylor expansion:

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

Solving for the critical point we obtain the Newton parameter update:

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

Q: Why is this bad for deep learning?
Second-Order Optimization

second-order Taylor expansion:

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

Solving for the critical point we obtain the Newton parameter update:

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

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q: Why is this bad for deep learning?
Second-Order Optimization

- Quasi-Newton methods (**BGFS** most popular): *instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).*

- **L-BFGS** (Limited memory BFGS): *Does not form/store the full inverse Hessian.*

\[ \theta^* = \theta_0 - H^{-1} \nabla \theta J(\theta_0) \]
L-BFGS

- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017
In practice:

- **AdamW** should probably be your “default” optimizer for new problems.
- **Adam** is a good second choice in many cases; it often works ok even with constant learning rate.
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule.
  - Try cosine schedule, very few hyperparameters!
Learning rate schedules
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Phases of learning...
Learning rate decays over time

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
Phases of learning...
Phases of learning...
Learning Rate Decay

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: \[
\alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos \left( \frac{t\pi}{T} \right) \right)
\]

$\alpha_0$ : Initial learning rate
$\alpha_t$ : Learning rate at epoch $t$
$T$ : Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**
\[
\alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos\left(\frac{t\pi}{T}\right) \right)
\]

- \(\alpha_0\) : Initial learning rate
- \(\alpha_t\) : Learning rate at epoch \(t\)
- \(T\) : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

- **Cosine:** $\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$
- **Linear:** $\alpha_t = \alpha_0 (1 - t/T)$

---

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

- **Cosine:** $\alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos \left( \frac{t\pi}{T} \right) \right)$
- **Linear:** $\alpha_t = \alpha_0 \left( 1 - \frac{t}{T} \right)$
- **Inverse sqrt:** $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$

$\alpha_0$ : Initial learning rate  
$\alpha_t$ : Learning rate at epoch $t$  
$T$ : Total number of epochs

Vaswani et al, “Attention is all you need”, NIPS 2017
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

Linear: $\alpha_t = \alpha_0 (1 - t/T)$

Inverse sqrt: $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$

Constant: $\alpha_t = \alpha_0$

Vaswani et al, “Attention is all you need”, NIPS 2017
High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this.

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N.

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017
From yesterday: with cosine and warmup
Improve test error
Beyond Training Error

Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?
Early Stopping: Always do this

Stop training the model when accuracy on the validation set decreases.
Or train for a long time, but always keep track of the model snapshot that worked best on val.
Model Ensembles

1. Train multiple independent models
2. At test time average their results
   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance
How to improve single-model performance?

Regularization
Regularization: Add term to loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W) \]

In common use:

- L2 regularization: \[ R(W) = \sum_k \sum_l W_{k,l}^2 \] (Weight decay)
- L1 regularization: \[ R(W) = \sum_k \sum_l |W_{k,l}| \]
- Elastic net (L1 + L2): \[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

Regularization: Dropout

\[ p = 0.5 \]  # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    """ X contains the data """

    # 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

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout
Regularization: Dropout

How can this possibly be a good idea?

- Forces the network to have a redundant representation;
- Prevents co-adaptation of features

Diagram:

- Nodes: has an ear, has a tail, is furry, has claws, mischievous look
- Connections: Cat score

Note: The diagram shows how dropout forces the network to consider alternative paths, leading to a redundant representation and preventing the co-adaptation of features.
Regularization: Dropout

How can this possibly be a good idea?

Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...
Dropout: Test time

Dropout makes our output random! \[ y = f_W(x, z) \]

Want to “average out” the randomness at test-time

\[ y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz \]

But this integral seems hard …
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz \]

Consider a single neuron.
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.

At test time we have:

\[ E[a] = w_1x + w_2y \]
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)\,dz \]

Consider a single neuron.

At test time we have:

\[ E[a] = w_1 x + w_2 y \]

During training we have:

\[ E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \]

\[ = \frac{1}{2}(w_1 x + w_2 y) \]
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.

At test time we have:

\[ E[a] = w_1x + w_2y \]

During training we have:

\[ E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) = \frac{1}{2}(w_1x + w_2y) \]

At test time, multiply by dropout probability
Dropout: Test time

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time
Vanilla Dropout: Not recommended implementation (see notes below)

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    # X contains the data

    # 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

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # 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
```

Dropout Summary

- **Drop in train time**
- **Scale at test time**
More common: “Inverted dropout”

```python
def train_step(X):
    # 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

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

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
```

test time is unchanged!
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

**Example:** Batch Normalization

**Training:**
Normalize using stats from random minibatches

**Testing:** Use fixed stats to normalize
Regularization: Data Augmentation

Load image and label

“cat”

Compute loss

CNN

This image by Nikita is licensed under CC-BY 2.0
Regularization: Data Augmentation

1. Load image and label
2. Transform image
3. "cat"
4. CNN
5. Compute loss
Data Augmentation
Horizontal Flips

![Cat Image]

![Flipped Cat Image]
Data Augmentation

Random crops and scales

**Training**: sample random crops / scales

ResNet:
1. Pick random $L$ in range $[256, 480]$
2. Resize training image, short side = $L$
3. Sample random $224 \times 224$ patch
Data Augmentation
Random crops and scales

**Training:** sample random crops / scales
ResNet:
1. Pick random $L$ in range $[256, 480]$
2. Resize training image, short side = $L$
3. Sample random $224 \times 224$ patch

**Testing:** average a fixed set of crops
ResNet:
1. Resize image at 5 scales: $\{224, 256, 384, 480, 640\}$
2. For each size, use 10 $224 \times 224$ crops: 4 corners + center, + flips
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set

2. Sample a “color offset” along principal component directions

1. Add offset to all pixels of a training image

(As seen in [Krizhevsky et al. 2012], ResNet, etc)
Data Augmentation

Get creative for your problem!

Examples of data augmentations:
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)
Automatic Data Augmentation

Cubuk et al., “AutoAugment: Learning Augmentation Strategies from Data”, CVPR 2019

Ranjay Krishna, Aditya Kusupati
Lecture 7 - 112
April 18, 2023
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
Regularization: DropConnect

**Training:** Drop connections between neurons (set weights to 0)

**Testing:** Use all the connections

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
**Regularization: Fractional Pooling**

**Training:** Use randomized pooling regions

**Testing:** Average predictions from several regions

**Examples:**
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling

Graham, “Fractional Max Pooling”, arXiv 2014
Regularization: Stochastic Depth

**Training**: Skip some layers in the network

**Testing**: Use all the layer

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth (will become more clear in next week's lecture)

Regularization: Cutout

**Training**: Set random image regions to zero

**Testing**: Use full image

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017

Works very well for small datasets like CIFAR, less common for large datasets like ImageNet
Regularization: Mixup

**Training**: Train on random blends of images

**Testing**: Use original images

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Target label: cat: 0.4, dog: 0.6

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018
Regularization: CutMix

**Training**: Train on random blends of images

**Testing**: Use original images

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup

Replace random crops of one image with another: e.g. 60% of pixels from cat, 40% from dog

Target label:
- cat: 0.4
- dog: 0.6

Yun et al, “CutMix: Regularization Strategies to Train Strong Classifiers with Localizable Features”, ICCV 2019
Regularization: Label Smoothing

Training: Change target distribution
Testing: Take argmax over predictions

Examples:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup
- Label Smoothing

Standard Training:
- Cat: 100%
- Dog: 0%
- Fish: 0%

Label Smoothing:
- Cat: 90%
- Dog: 5%
- Fish: 5%

Set target distribution to be $1 - \frac{K-1}{K} \epsilon$ on the correct category and $\epsilon/K$ on all other categories, with $K$ categories and $\epsilon \in [0, 1]$. Loss is cross-entropy between predicted and target distribution.

Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”, CVPR 2015
Regularization - In practice

**Training**: Add random noise

**Testing**: Marginalize over the noise

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup

- Use dropout for large fully-connected layers
- Using batchnorm is always a good idea
- Try Cutout, MixUp, CutMix, Stochastic Depth, Label Smoothing to squeeze out a bit of extra performance
Choosing Hyperparameters
Choosing Hyperparameters: Grid Search

Choose several values for each hyperparameter
(Of often space choices log-linearly)

Example:
Weight decay: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$
Learning rate: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$

Evaluate all possible choices on this hyperparameter grid
Choosing Hyperparameters: Random search

Choose several values for each hyperparameter
(Often space choices log-linearly)

Example:
Weight decay: \textbf{log-uniform} on \([1 \times 10^{-4}, 1 \times 10^{-1}]\)
Learning rate: \textbf{log-uniform} on \([1 \times 10^{-4}, 1 \times 10^{-1}]\)

Run many different trials
Random Search vs. Grid Search

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017
Choosing Hyperparameters
(without tons of GPUs)
Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. log(C) for softmax with C classes
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization
Loss explodes toInf or NaN? LR too high, bad initialization
Choosing Hyperparameters

**Step 1:** Check initial loss

**Step 2:** Overfit a small sample

**Step 3:** Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: $1e^{-1}$, $1e^{-2}$, $1e^{-3}$, $1e^{-4}$
Choosing Hyperparameters

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down
**Step 4:** Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0
Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
**Step 3**: Find LR that makes loss go down
**Step 4**: Coarse grid, train for ~1-5 epochs
**Step 5**: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss and accuracy curves
Q1. You see this. What should you do?
Q2. You see this. What should you do?
Q3. You see this. What should you do?
Look at learning curves!

Losses may be noisy, use a scatter plot and also plot moving average to see trends better.
Q4. You see this. What should you do?
Q5. You see this. What should you do?
Q5. You see this. What should you do?
Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters.

check out weights and biases
You can plot all your loss curves for different hyperparameters on a single plot.
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss and accuracy curves
Step 7: GOTO step 5
Hyperparameters to play with:
- learning rate,
- Its decay schedule, update type
- regularization (L2/Dropout strength)
Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules

- Improve your test error:
  - Regularization
  - Choosing Hyperparameters
Next time: Visualizing and understanding neural networks
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al., “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Cyclic learning rate schedules can make this work even better!

Huang et al. "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```python
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Track the ratio of weight updates / weight magnitudes:

```python
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so