Topics:

• Training Neural Networks (Part 2)
Administrative

- Project Proposal deadline **postponed to Oct 3\(^{rd}\) (Monday)**
  - No grace period!
- Google cloud coupon instruction released on Piazza
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**

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

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

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. \(\exp()\) is a bit compute expensive
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

\[ f(x) = \max(\alpha x, x) \]

backprop into \( \alpha \) (parameter)

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Exponential Linear Units (ELU)

- All benefits of ReLU
- Negative saturation encodes presence of features (all goes to $\alpha$), not magnitude
- Same in backprop
- Compared with Leaky ReLU: more robust to noise

$$f(x) = \begin{cases} 
    x & \text{if } x > 0 \\
    \alpha (\exp(x) - 1) & \text{if } x \leq 0 
\end{cases}$$

(Alpha default = 1)
Activation Functions

Scaled Exponential Linear Units (SELU)

- Scaled version of ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

\[ f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda \alpha (e^x - 1) & \text{otherwise} \end{cases} \]

\[ \alpha = 1.6732632423543772848170429916717 \]
\[ \lambda = 1.0507009873554804934193349852946 \]

Derivation takes 91 pages of math in appendix...

(Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017)
TLDR: In practice:

- Many possible choices beyond what we’ve talked here, but …
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / ELU / SELU
  - To squeeze out some marginal gains
- Don’t use sigmoid or tanh
Data Preprocessing
Data Preprocessing

(Assume $X$ [NxD] is data matrix, each example in a row)

$$X \leftarrow \text{np.mean}(X, \text{axis} = 0)$$

$$X \leftarrow \text{np.std}(X, \text{axis} = 0)$$
Data Preprocessing

**Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize

**After normalization**: less sensitive to small changes in weights; easier to optimize
Weight Initialization
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Forward pass for a 6-layer net with hidden size 4096
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

All activations tend to zero for deeper network layers

**Q:** What do the gradients \( \frac{dL}{dW} \) look like?

**Hint:** \[
\frac{\partial L}{\partial w} = x^T \left( \frac{\partial L}{\partial y} \right)
\]
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  
hs = []  
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

All activations saturate

Q: What do the gradients look like?

More generally, gradient explosion.
Weight Initialization: “Xavier” Initialization

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7  
hs = []  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is `filter_size^2 * input_channels`

Let: \( y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din} \)

Assume: \( \text{Var}(x_1) = \text{Var}(x_2) = \ldots = \text{Var}(x_{Din}) \)

We want: \( \text{Var}(y) = \text{Var}(x_i) \)

\[
\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}) \\
= \text{Din} \text{Var}(x_i w_i) \\
= \text{Din} \text{Var}(x_i) \text{Var}(w_i)
\]

[Assume all \( x_i, w_i \) are iid]

So, \( \text{Var}(y) = \text{Var}(x_i) \) only when \( \text{Var}(w_i) = 1/\text{Din} \)

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010
Weight Initialization: Kaiming / MSRA Initialization

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Issue: Half of the activation get killed.
Solution: make the non-zero output variance twice as large as input

This Time:

**Training** Deep Neural Networks
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble
Batch Normalization
Recall: Normalization

Problem: Can’t do this for intermediate layers! Need fixed statistics (e.g., mean & std), but activations change as the training progresses.
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations $x$ at some layer. To make each dimension zero-mean unit-variance, apply:

$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x]}}$$

this is a vanilla differentiable function...
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

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

**Input:** $x : N \times D$

**Per-channel mean, shape is D**

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

**Per-channel var, shape is D**

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

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

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

(Prevent div by 0 err)

[ioffe and Szegedy, 2015]
Batch Normalization

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

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

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

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

Problem: What if zero-mean, unit variance is too hard of a constraint? E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime

[ioffe and Szegedy, 2015]
Batch Normalization

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

Learnable scale and shift parameters: \( \gamma, \beta : \mathbb{R}^D \)

We want to give the model a chance to adjust batchnorm if the default is not optimal. Learning \( \gamma = \sigma \) and \( \beta = \mu \) will recover the identity function!

Per-channel mean, shape is D

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

Per-channel var, shape is D

\[
\sigma^2_j = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

Normalized x, Shape is \( N \times D \)

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

Output, Shape is \( N \times D \)

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

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

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

Learnable scale and shift parameters:
\( \gamma, \beta : \mathbb{R}^D \)

Activations become fixed after training. Can calculate training set-wide statistics for inference-time normalization.

Do moving average to save compute.

Estimates depend on minibatch; can’t do this at test-time!

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

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

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

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}
\]
Batch Normalization: Test-Time

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

Learnable scale and shift parameters:
\[ \gamma, \beta : \mathbb{R}^D \]

Per-channel mean, shape is D
\[ \mu_j = \text{(Moving) average of values seen during training} \]

Per-channel var, shape is D
\[ \sigma_j^2 = \text{(Moving) average of values seen during training} \]

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

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

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer
Q: Should you put batchnorm before or after ReLU?
A: Topic of debate. Original paper says BN->ReLU. Now most commonly ReLU->BN. If BN-> ReLU and zero mean, ReLU kills half of the activations, but in practice makes insignificant differences.

Q: Should you normalize the input (e.g., images) with batchnorm?
A: No, you already have the fixed & correct dataset statistics, no need to do batchnorm.

Q: How many parameters does a batchnorm layer have?
A: Input dimension * 4: beta, gamma, moving average mu, moving average sigma. Only beta and gamma are trainable parameters.
Batch Normalization

- Makes deep networks **much** easier to train!
  - If you are interested in the theory, read https://arxiv.org/abs/1805.11604
  - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats
Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

\[
x: \ N \times D
\]

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

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

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

Batch Normalization for **convolutional** networks

(Spatial Batchnorm, BatchNorm2D)

\[
x: \ N \times C \times H \times W
\]

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

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

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

Batch Normalization for fully-connected networks

\[ x : N \times D \]

Normalize

\( \mu, \sigma : 1 \times D \)

\( \gamma, \beta : 1 \times D \)

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


Layer Normalization for fully-connected networks

Same behavior at train and test!

\[ x : N \times D \]

Normalize

\( \mu, \sigma : N \times 1 \)

\( \gamma, \beta : 1 \times D \)

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

More flexible (can use \( N = 1! \)), works well with sequence models (RNN, Transformers)
Instance Normalization

**Batch Normalization** for convolutional networks

\[
\begin{align*}
    x & : \mathbb{N} \times C \times H \times W \\
    \mu, \sigma & : 1 \times C \times 1 \times 1 \\
    \gamma, \beta & : 1 \times C \times 1 \times 1 \\
    y & = \gamma (x - \mu) / \sigma + \beta
\end{align*}
\]

**Instance Normalization** for convolutional networks

Same behavior at train / test!

\[
\begin{align*}
    x & : \mathbb{N} \times C \times H \times W \\
    \mu, \sigma & : \mathbb{N} \times C \times 1 \times 1 \\
    \gamma, \beta & : 1 \times C \times 1 \times 1 \\
    y & = \gamma (x - \mu) / \sigma + \beta
\end{align*}
\]

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
(Fancier) Optimizers
Optimization

# 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

- Stochastic minibatch gives a noisy estimate of the true gradient direction. Very problematic when the batch size is small (e.g., due to compute resource limit).
- Poorly-selected learning rate makes the oscillation worse (overshoot)

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
SGD + Momentum

Intuitions:
• Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
• Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
• Momentum can “push” the ball over saddle points or local minima.
SGD + Momentum

Intuitions:
• Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
• Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
• Momentum can “push” the ball over saddle points or local minima.
SGD: the simple two line update code

SGD

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

```
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

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

- 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

### SGD

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

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

### SGD+Momentum

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

- \( v_0 = 0 \)

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

while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x = learning_rate * vx

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

**SGD+Momentum**

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

**SGD+Momentum**

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

\[\text{vx} = 0\]

**while** True:

\[dx = \text{compute\_gradient}(x)\]
\[vx = \rho \times vx - \text{learning\_rate} \times dx\]
\[x += vx\]

\[\text{vx} = 0\]

**while** True:

\[dx = \text{compute\_gradient}(x)\]
\[vx = \rho \times vx + dx\]
\[x -= \text{learning\_rate} \times vx\]

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**

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

“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
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
Optimization: Problem #3 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

Long, narrow ravines:

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

AdaGrad

```
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
AdaGrad

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.
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)
```

Q2: What happens to the step size over long time?
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)
```

Q2: What happens to the step size over long time? Decays to zero
RMSProp: “Leaky AdaGrad”

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)
```

Sort of like RMSProp with momentum

Q: What happens at first timestep?

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
    firstUnbias = firstMoment / (1 - beta1 ** t)
    secondUnbias = secondMoment / (1 - beta2 ** t)
    x -= learning_rate * firstUnbias / (np.sqrt(secondUnbias) + 1e-7)
```

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

Momentum
AdaGrad / RMSProp

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

Adam (full form)

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

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
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.
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.
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 at 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:**

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

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

**Linear:**
\[ \alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right) \]

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

---

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

$\alpha_0$ : 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
First-Order Optimization

Loss

w1
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 \( O(N^3) \)
\( N = \) 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.
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:

- **Adam** is a good default 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!

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)
Regularization
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

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

X X X

cat score
Regularization: Dropout

How can this possibly be a good idea?

Another interpretation:

Dropout is training a large \textit{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!

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

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

During training we have:
\[
\begin{align*}
E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\
&\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\
&= \frac{1}{2}(w_1x + w_2y)
\end{align*}
\]
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
Dropout Summary

Dropout Summary

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

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
```

*drop in train time*

*scale at test time*
More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

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
Next Time:

**Training Deep Neural Networks**
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble