Topics:
- Training Neural Networks 2

Zsolt Kira
Georgia Tech
• PS3/HW3 – Will be out today
  – Due in two weeks

• Submit project proposal by Thursday (and notify me)
Previously: 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**
\[
\begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0
\end{cases}
\]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Previously: 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**
  \[ \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
  \end{cases} \]
Previously: 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
Previously: Data Preprocessing
Previously: Babysitting Learning
Previously: Hyperparameter Search

Grid Layout

Random Layout

Coarse to fine search

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Today

- Fancier optimization
- Regularization
- Transfer Learning
Optimization

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

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

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large
Optimization: Problems with SGD (1)

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
Optimization: Problems with SGD (2)

What if the loss function has a local minima or saddle point?
Optimization: Problems with SGD (2)

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

Zero gradient, gradient descent gets stuck
Optimization: Problems with SGD (2)

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: Problems with SGD (3)

Our gradients come from minibatches so they can be noisy!

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
SGD + Momentum

**SGD**

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

```python
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```

**SGD+Momentum**

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

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

```python
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

- 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

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

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
SGD + Momentum

Local Minima    Saddle points

Poor Conditioning

Gradient Noise

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
**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 Momentum

“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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

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

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Nesterov Momentum

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

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

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

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

\[
\begin{align*}
\text{dx} &= \text{compute\_gradient}(x) \\
\text{old\_v} &= v \\
\text{v} &= \text{rho} \ast v - \text{learning\_rate} \ast \text{dx} \\
\text{x} &= \text{rho} \ast \text{old\_v} + (1 + \text{rho}) \ast v
\end{align*}
\]
Nesterov Momentum

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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
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

Q: What happens with AdaGrad?

Progress along "steep" directions is damped; progress along "flat" directions is accelerated

slide credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
AdaGrad

```
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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
RMSProp

AdaGrad

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

RMSProp

```
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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
RMSProp

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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^2
    x = learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Adam (almost)

```
f_{\text{first\_moment}} = 0
s_{\text{second\_moment}} = 0
while True:
    dx = compute\_gradient(x)
    f_{\text{first\_moment}} = beta1 \cdot f_{\text{first\_moment}} + (1 - beta1) \cdot dx
    s_{\text{second\_moment}} = beta2 \cdot s_{\text{second\_moment}} + (1 - beta2) \cdot dx \cdot dx
    x \leftarrow \text{learning\_rate} \cdot f_{\text{first\_moment}} / (\sqrt{s_{\text{second\_moment}}} + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, “Adam: A method for stochastic optimization”,
ICLR 2015
Adam (full form)

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


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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 with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Adam

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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.

=> **Learning rate decay over time!**

**step decay:**
e.g. decay learning rate by half every few epochs.

**exponential decay:**
\[ \alpha = \alpha_0 e^{-kt} \]

**1/t decay:**
\[ \alpha = \frac{\alpha_0}{1 + kt} \]
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
First-Order Optimization
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: What is nice about this update?
Second-Order Optimization

second-order Taylor expansion:

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

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

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

Q: What is nice about this update?
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:

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

No hyperparameters!
No learning rate!
(Though you might use one in practice)

Q: What is nice about this update?
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:

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

Q2: 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 \(O(N^3)\)
\(N = \text{(Tens or Hundreds of) Millions}\)

Q2: Why is this bad for deep learning?
Second-Order Optimization

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

- 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
- **SGD+Momentum** with learning rate decay often outperforms Adam by a bit, but requires more tuning

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

Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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
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
```

Early Stopping

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
How to improve single-model performance?

Regularization

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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!

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

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

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

At test time, multiply by dropout probability

Note: Here, dropout probability means the probability of keeping an activation – sometimes people define this as the opposite...
Dropout: Test time

```python
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
```

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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 forward pass
- scale at test time
Alternative: “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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Regularization: Data Augmentation

Load image and label → “cat” → CNN → Compute loss

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Regularization: Data Augmentation

Load image and label → "cat" → Transform image → CNN → Compute loss

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Data Augmentation
Horizontal Flips

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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 x 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 x 224 crops: 4 corners + center, + flips
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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!

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)
Regularization: A common pattern

**Training:** Add random noise
**Testing:** Marginalize over the noise

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

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

Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

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

Graham, “Fractional Max Pooling”, arXiv 2014
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

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

Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016
Transfer Learning

“You need a lot of data if you want to train/use CNNs”
Transfer Learning

“You need a lot of data if you want to train/use CNNs”
Transfer Learning with CNNs

1. Train on Imagenet


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Transfer Learning with CNNs

1. Train on Imagenet

2. Small Dataset (C classes)
   - Reinitialize this and train
   - Freeze these

3. Bigger dataset
   - Train these
   - With bigger dataset, train more layers
   - Fine-tune these
   - Lower learning rate when finetuning; 1/10 of original LR is good starting point

---

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

More specific

More generic

<table>
<thead>
<tr>
<th></th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>More specific</td>
<td>More generic</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td><strong>FC-1000</strong></td>
<td><strong>FC-1000</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>FC-4096</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>FC-4096</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-612</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-612</strong></td>
<td><strong>Conv-612</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-256</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-256</strong></td>
<td><strong>Conv-256</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-128</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-128</strong></td>
<td><strong>Conv-128</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-64</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conv-64</strong></td>
<td><strong>Conv-64</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Image</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>very similar</td>
<td></td>
</tr>
<tr>
<td>very different</td>
<td></td>
</tr>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
</tr>
</tbody>
</table>

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
### Table: Data and Model Performance

<table>
<thead>
<tr>
<th>Data Quantity</th>
<th>Similar Dataset</th>
<th>Different Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very little data</td>
<td>Use Linear Classifier on top layer</td>
<td>You're in trouble... Try linear classifier from different stages</td>
</tr>
<tr>
<td>Quite a lot of data</td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers</td>
</tr>
</tbody>
</table>

**Diagram:**

- More specific
- More generic

---

*Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

- Object Detection (Fast R-CNN)
  - CNN pretrained on ImageNet
  - Word vectors pretrained with word2vec

- Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
The most effective method: Gather more data!

Revisiting the Unreasonable Effectiveness of Data

Deep Learning Scaling is Predictable, Empirically
Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianninejad, Md. Mostofa Ali Patwary, Yang Yang, Yanqi Zhou
Takeaway for your projects and beyond:
Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Life is never so simple

There are several areas being researched
- Batch size
- Regularization and generalization
- Overparameterization and why SGD is so good

Why is this still not understood?
- Our understanding comes from built-in intuition that is repeated but not always tested
- Difficult to apply theory being developed
DNNs: Over-parametrization yields great generalization even without any explicit regularization

Note the zero training error in the over-parametrized part
• DNN forms patterns, thus generalizing well
• It also memorizes noisy examples, but in a harmless way
• All these need more understanding

The interesting part – Great generalization!

Zhang et al, Theory of deep learning III, September 2017

Slide Credit: S. Sathiya Keerthi
More examples of great generalization without any regularization

- \( n \) = number of examples
- \( p \) = number of parameters
- \( d \) = number of inputs
- \( k \) = number of layers

Note that networks with larger \( p/n \) have better generalization

<table>
<thead>
<tr>
<th>Model</th>
<th>parameters</th>
<th>( p/n )</th>
<th>Train loss</th>
<th>Test error</th>
</tr>
</thead>
<tbody>
<tr>
<td>CudaConvNet</td>
<td>145,578</td>
<td>2.9</td>
<td>0</td>
<td>23%</td>
</tr>
<tr>
<td>CudaConvNet (with regularization)</td>
<td>145,578</td>
<td>2.9</td>
<td>0.34</td>
<td>18%</td>
</tr>
<tr>
<td>MicroInception</td>
<td>1,649,402</td>
<td>33</td>
<td>0</td>
<td>14%</td>
</tr>
<tr>
<td>ResNet</td>
<td>2,401,440</td>
<td>48</td>
<td>0</td>
<td>13%</td>
</tr>
</tbody>
</table>

What happens when I turn off the regularizers?

CIFAR10

\( n=50,000 \)
\( d=3,072 \)
\( k=10 \)

Slide Credit: S. Sathiya Keerthi

Ben Recht Talk slides, ICLR 2017
History of powerful DNN solvers on ImageNet (15 million examples) [Ref]
Batch Size

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima
Nitish Shirish, Keskar, Dheevatsa, Mudigere, Jorge, Nocedal, Mikhail Smelyanskiy, Ping Tak, Peter Tang
Large batch yields inferior generalization..

Keskar et al: This is due to flatness properties of solutions

This experiment was done on a modified AlexNet (CNN) on CIFAR-10 dataset

Slide Credit: S. Sathiya Keerthi
Generalization (negatively) correlates well with sharpness, thus explaining the superiority of small batch over large batch.

This experiment was done on a modified AlexNet (CNN) on CIFAR-10 dataset.

Slide Credit: S. Sathiya Keerthi
Why flatness means better generalization?

Flatness implies that the test loss will be close to the training loss

Slide Credit: S. Sathiya Keerthi
One solution: Distributed SGD

Slide Credit: Dimitris Papailiopoulos