Lecture 6:

Training Neural Networks, Part 2

Administrative

A2 is out. It's meaty. It's due Feb 5 (next Friday)

You'll implement:

- Neural Nets (with Layer Forward/Backward API)
- Batch Norm
- Dropout
- ConvNets

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Activation Functions

Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$

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

Data Preprocessing

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)

Weight Initialization

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Babysitting the learning process

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Cross-validation

TODO

- Parameter update schemes
- Learning rate schedules
- Dropout
- Gradient checking
- Model ensembles

Parameter Updates

Training a neural network, main loop:

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Training a neural network, main loop:

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

Image credits: Alec Radford

Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:

It is bouncing and is very slow to converge

Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

Momentum update

```
# Gradient descent update
x += - learning_rate * dx
```

Take the grad we computed,
Building v in an exponential way. miu here is a happy parameter, (0-1). used for decaying the previous a. v (velocity) is accelerated by force.
For the shallow one, it slowly accelerate.
For the steep one, you are being pulled toward the center.
Very Intuitive
v = 0 initially

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update

```
# Gradient descent update
x += - learning_rate * dx
```

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD vs Momentum

notice momentum overshooting the target, but overall getting to the minimum much faster.

Nesterov Momentum update

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Ordinary momentum update:

Nesterov Momentum update

Momentum update

Nesterov momentum update

v is built in two parts:
built up some momentum step (green, former)
some grad step (red, latter)
Blue is the final position.
For Nest one, we know we are going to reach the blue point,
we do a look ahead: calculate the grad at blue point rather
than the start point

Nesterov Momentum update

Momentum update

Nesterov momentum update

Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day: $\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$

Nesterov Momentum update

Always assures a better performance than the former one

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\boxed{\theta_{t-1} + \mu v_{t-1}})$$

green one

$$\theta_t = \theta_{t-1} + v_t$$

grad. at blue point

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Theta is gradient para representing the grad at that point. Theta(t) used be only be relavant to d(Theta t), not it is relevant to d(Theta t-1)

Variable transform and rearranging saves the day:

$$\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$$

Replace all thetas with phis, rearrange and obtain:

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```

nag = Nesterov Accelerated Gradient

AdaGrad update

[Duchi et al., 2011]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

AdaGrad update

scaling the gradient. cache is positive only.
Cache here is a giant vector of the same size as your para vector. The operation here is element wise
In every dimension we keep tracking the sum of the grad.
Everything dimension has own learning rate in terms of what scale of grad you've seen

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

Problem: learning rate decades to 0

Q: What happens with AdaGrad? vertical (steep): large, and divided, slower. vice versa

AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

RMSProp update

[Tieleman and Hinton, 2012]

Improvement on Ada

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

We have the decay rate, so the cache is leaking, learning rate is not going to 0

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1 \left(\partial E / \partial w (t) \right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1 \left(\partial E / \partial w (t) \right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

adagrad
rmsprop

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Keep tracking of moment in a decaying way and in the mean time scaling it

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]

t is basically compensating that m and v are initialized 0. it scales up m and v in first few iterations so you dont end up with a biased estimate of first moment and second moment.

```
# Adam
m,v = #... initialize caches to zeros
for t in xrange(1, big_number):
  dx = # ... evaluate gradient
  m = beta1*m + (1-beta1)*dx # update first moment
  v = beta2*v + (1-beta2)*(dx**2) # update second moment
  mb = m/(1-beta1**t) # correct bias
  vb = v/(1-beta2**t) # correct bias
  x += - learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

momentum

bias correction

(only relevant in first few iterations when t is small)

RMSProp-like

The bias correction compensates for the fact that m,v are initialized at zero and need some time to “warm up”.

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 rates is best to use?
Use high rate first, some point decay your learning rate
(ways of decaying below)

=> 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:

$$\alpha = \alpha_0 / (1 + kt)$$

Second order optimization methods

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Second order methods for optimization: they end up forming a larger approximation for your lost function so they dont only approximated with this basically hpyerplane but also approximated hessian that tells you how your surface is curving so you dont only need the grad you also need the hessian (compute that)

Cons: Say 1 billion para, you have 1b * 1b Hessian

Q: what is nice about this update?

no hyperparameters! (e.g. learning rate). No need for learning. you know the grad and curvature, so you know exactly where to go

Second order optimization methods

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

notice:
no hyperparameters! (e.g. learning rate)

Q2: why is this impractical for training Deep Neural Nets?

Second order optimization methods

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

BGFS: Get away with not inverting the Hessian, it builds up an approximation through successive updates that are all ranked one (vector)

- 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).

Solved the memory problem for BFGS

- **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 No Stochastic noise

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

In practice:

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

Evaluation:

Model Ensembles

Refer to TTIC Architecture

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.

Fun Tips/Tricks:

X_test is the running sum of actual vector and decaying. This performs slightly better. You are stepping around minimum, it takes the average so a little bit closer

- can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:

```
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
```

Regularization **(dropout)**

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"

(a) Standard Neural Net (b) After applying dropout.

[Srivastava et al., 2014]

```python
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)
```

During forward pass, On possibility (1-p) of setting neurons to 0

Have to adjust back propagate too, you back propagated by multiplying by u2 and u1

Example forward pass with a 3-layer network using dropout

Waaaait a second…
How could this possibly be a good idea?

Waaaait a second…
How could this possibly be a good idea?

Forces the network to have a redundant representation.

Waaaait a second… How could this possibly be a good idea?

If one neuron is dropped, it gets zero in forward propagation and it's grad is not affected in backprapagation. So its not updated. Kind of training a subset of neuron network (but all neurons share the same para)

Another interpretation:

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

Each binary mask is one model, gets trained on only ~one datapoint.

At test time….

The noise: possibility

BGFS: Get away with not inverting the Hessian, it builds up an approximation through successive updates that are all ranked one (vector)

Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

In the forward with test images we are not going to drop out any units. Be careful because during train is different

(this can be shown to be an approximation to evaluating the whole ensemble)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

Q: Suppose that with all inputs present at test time the output of this neuron is x.

What would its output be during training time, in expectation? (e.g. if p = 0.5)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

during test: **a = w0*x + w1*y**

during train:

E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= ½ * **(w0*x + w1*y)**

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

during test: **a = w0*x + w1*y**

during train:

$$
\begin{aligned}
E[a] = \tfrac{1}{4} * (&w0*0 + w1*0 \\
&w0*0 + w1*y \\
&w0*x + w1*0 \\
&w0*x + w1*y) \\
= \tfrac{1}{4} * (&2\, w0*x + 2\, w1*y) \\
= \tfrac{1}{2} * (&\mathbf{w0*x + w1*y})
\end{aligned}
$$

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

Need to compensate for that. Scale your activation function

We can do something approximate analytically

```
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:
<u>output at test time</u> = <u>expected output at training time</u>

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
```

This is scaling in test time

drop in forward pass

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

scale at test time

More common: “Inverted dropout”

```
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
```

However, to improve performance, we can also scale at training time, same result, but test time is unchanged

test time is unchanged!

<fun story time>

(Deep Learning Summer School 2012)

Gradient Checking

(see class notes...)

fun guaranteed.

Skipped

Gradient Checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient. In practice, the process is much more involved and error prone. Here are some tips, tricks, and issues to watch out for:

Use the centered formula. The formula you may have seen for the finite difference approximation when evaluating the numerical gradient looks as follows:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

where h is a very small number, in practice approximately 1e-5 or so. In practice, it turns out that it is much better to use the *centered* difference formula of the form:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

This requires you to evaluate the loss function twice to check every single dimension of the gradient (so it is about 2 times as expensive), but the gradient approximation turns out to be much more precise. To see this, you can use Taylor expansion of $f(x+h)$ and $f(x-h)$ and verify that the first formula has an error on order of $O(h)$, while the second formula only has error terms on order of $O(h^2)$ (i.e. it is a second order approximation).

Use relative error for the comparison. What are the details of comparing the numerical gradient f'_n and analytic gradient f'_a? That is, how do we know if the two are not compatible? You might be temped to keep track of the difference $|f'_a - f'_n|$ or its square and define the gradient check as failed if that difference is above a threshold. However, this is problematic. For example, consider the case where their difference is 1e-4. This seems like a very appropriate difference if the two gradients are about 1.0, so we'd consider the two gradients to match. But if the gradients were both on order of 1e-5 or lower, then we'd consider 1e-4 to be a huge difference and likely a failure. Hence, it is always more appropriate to consider the *relative error*:

$$\frac{|f'_a - f'_n|}{\max(|f'_a|, |f'_n|)}$$

which considers their ratio of the differences to the ratio of the absolute values of both gradients. Notice that normally the relative error formula only includes one of the two terms (either one), but I prefer to max (or add) both to make it symmetric and to prevent dividing by zero in the case where one of the two is zero (which can often happen, especially with ReLUs). However, one must explicitly keep track of the case where both are zero and pass the gradient check in that edge case. In practice:

- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable
- 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.
- 1e-7 and less you should be happy.

Also keep in mind that the deeper the network, the higher the relative errors will be. So if you are gradient checking the input data for a 10-layer network, a relative error of 1e-2 might be okay because the errors build up on the way. Conversely, an error of 1e-2 for a single differentiable function likely indicates incorrect gradient.

Use double precision. A common pitfall is using single precision floating point to compute gradient check. It is often that case that you might get high relative errors (as high as 1e-2) even with a correct gradient implementation. In my experience I've sometimes seen my relative errors plummet from 1e-2 to 1e-8 by switching to double precision.

Stick around active range of floating point. It's a good idea to read through "What Every Computer Scientist Should Know About Floating-Point Arithmetic", as it may demystify your errors and enable you to write more careful code. For example, in neural nets it can be common to normalize the loss function over the batch. However, if your gradients per datapoint are very small, then *additionally* dividing them by the number of data points is starting to give very small numbers, which in turn will lead to more numerical issues. This is why I like to always print the raw numerical/analytic gradient, and make sure that the numbers you are comparing are not extremely small (e.g. roughly 1e-10 and smaller in absolute value is worrying). If they are you may want to temporarily scale your loss function up by a constant to bring them to a "nicer" range where floats are more dense - ideally on the order of 1.0, where your float exponent is 0.

Kinks in the objective. One source of inaccuracy to be aware of during gradient checking is the problem of *kinks*. Kinks refer to non-differentiable parts of an objective function, introduced by functions such as ReLU ($max(0,x)$), or the SVM loss, Maxout neurons, etc. Consider gradient checking the ReLU function at $x = -1e6$. Since $x < 0$, the analytic gradient at this point is exactly zero. However, the numerical gradient would suddenly compute a non-zero gradient because $f(x+h)$ might cross over the kink (e.g. if $h > 1e-6$) and introduce a non-zero contribution. You might think that this is a pathological case, but in fact this case can be very common. For example, an SVM for CIFAR-10 contains up to 450,000 $max(0,x)$ terms because there are 50,000 examples and each example yields 9 terms to the objective. Moreover, a Neural Network with an SVM classifier will contain many more kinks due to ReLUs.

Note that it is possible to know if a kink was crossed in the evaluation of the loss. This can be done by keeping track of the identities of all "winners" in a function of form $max(x,y)$; That is, was x or y higher during the forward pass. If the identity of at least one winner changes when evaluating $f(x+h)$ and then $f(x-h)$, then a kink was crossed and the numerical gradient will not be exact.

Use only few datapoints. One fix to the above problem of kinks is to use fewer datapoints, since loss functions that contain kinks (e.g. due to use of ReLUs or margin losses etc.) will have fewer kinks with fewer datapoints, so it is less likely for you to cross one when you perform the finite different approximation. Moreover, if your gradcheck for only ~2 or 3 datapoints then you would almost certainly gradcheck for an entire batch. Using very few datapoints also makes your gradient check faster and more efficient.

Be careful with the step size h. It is not necessarily the case that smaller is better, because when h is much smaller, you may start running into numerical precision problems. Sometimes when the gradient doesn't check, it is possible that you change h to be 1e-4 or 1e-6 and suddenly the gradient will be correct. This wikipedia article contains a chart that plots the value of **h** on the x-axis and the numerical gradient error on the y-axis.

Gradcheck during a "characteristic" mode of operation. It is important to realize that a gradient check is performed at a particular (and usually random), single point in the space of parameters. Even if the gradient check succeeds at that point, it is not immediately certain that the gradient is correctly implemented globally. Additionally, a random initialization might not be the most "characteristic" point in the space of parameters and may in fact introduce pathological situations where the gradient seems to be correctly implemented but isn't. For instance, an SVM with very small weight initialization will assign almost exactly zero scores to all datapoints and the gradients will exhibit a particular pattern across all datapoints. An incorrect implementation of the gradient could still produce this pattern and not generalize to a more characteristic mode of operation where some scores are larger than others. Therefore, to be safe it is best to use a short **burn-in** time during which the network is allowed to learn and perform the gradient check after the loss starts to go down. The danger of performing it at the first iteration is that this could introduce pathological edge cases and mask an incorrect implementation of the gradient.

Don't let the regularization overwhelm the data. It is often the case that a loss function is a sum of the data loss and the regularization loss (e.g. L2 penalty on weights). One danger to be aware of is that the regularization loss may overwhelm the data loss, in which case the gradients will be primarily coming from the regularization term (which usually has a much simpler gradient expression). This can mask an incorrect implementation of the data loss gradient. Therefore, it is recommended to turn off regularization and check the data loss alone first, and then the regularization term second and independently. One way to perform the latter is to hack the code to remove the data loss contribution. Another way is to increase the regularization strength so as to ensure that its effect is non-negligible in the gradient check, and that an incorrect implementation would be spotted.

Remember to turn off dropout/augmentations. When performing gradient check, remember to turn off any non-deterministic effects in the network, such as dropout, random data augmentations, etc. Otherwise these can clearly introduce huge errors when estimating the numerical gradient. The downside of turning off these effects is that you wouldn't be gradient checking them (e.g. it might be that dropout isn't backpropagated correctly). Therefore, a better solution might be to force a particular random seed before evaluating both $f(x+h)$ and $f(x-h)$, and when evaluating the analytic gradient.

Check only few dimensions. In practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient and assume that the others are correct. **Be careful**: One issue to be careful with is to make sure to gradient check a few dimensions for every separate parameter. In some applications, people combine the parameters into a single large parameter vector for convenience. In these cases, for example, the biases could only take up a tiny number of parameters from the whole vector, so it is important to not sample at random but to take this into account and check that all parameters receive the correct gradients.

Convolutional Neural Networks

[LeNet-5, LeCun 1980]

A bit of history:

Hubel & Wiesel,

1959

RECEPTIVE FIELDS OF SINGLE NEURONES IN
THE CAT'S STRIATE CORTEX

1962

RECEPTIVE FIELDS, BINOCULAR INTERACTION
AND FUNCTIONAL ARCHITECTURE IN THE CAT'S VISUAL CORTEX

1968...

Video time https://youtu.be/8VdFf3egwfg?t=1m10s

A bit of history

Topographical mapping in the cortex: nearby cells in cortex represented nearby regions in the visual field

Hierarchical organization

A bit of history:

Neurocognitron
[Fukushima 1980]

"sandwich" architecture (SCSCSC…)
simple cells: modifiable parameters
complex cells: perform pooling

A bit of history:

Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]

LeNet-5

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

IMAGENET

"AlexNet"

Fast-forward to today: ConvNets are everywhere

Classification

Retrieval

[Krizhevsky 2012]

Fast-forward to today: ConvNets are everywhere

Detection

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation

[Farabet et al., 2012]

Fast-forward to today: ConvNets are everywhere

self-driving cars

NVIDIA Tegra X1

Fast-forward to today: ConvNets are everywhere

[Taigman et al. 2014]

[Simonyan et al. 2014]

[Goodfellow 2014]

Fast-forward to today: ConvNets are everywhere

[Toshev, Szegedy 2014]

[Mnih 2013]

Fast-forward to today: ConvNets are everywhere

[Ciresan et al. 2013]

[Sermanet et al. 2011]
[Ciresan et al.]

Fast-forward to today: ConvNets are everywhere

[Turaga et al., 2010]

I caught this movie on the Sci-Fi channel recently. It actually turned out to be pretty decent as far as B-list horror/suspense films go. **Two guys (one naive and one loud mouthed a **) take a road trip to stop a wedding but have the worst possible luck when a maniac in a freaky, make-shift tank/truck hybrid decides to play cat-and-mouse with them.** Things are further complicated when they pick up a ridiculously whorish hitchhiker. What makes this film unique is that the combination of comedy and terror actually work in this movie, unlike so many others. The two guys are likable enough and there are some good chase/suspense scenes. Nice pacing and comic timing make this movie more than passable for the horror/slasher buff. **Delinitely worth checking out.**

I just saw this on a local independent station in the New York City area. **The cast showed promise but when I saw the director, George Cosmotos, I became suspicious. And sure enough, it was every bit as bad, every bit as pointless and stupid as every George Cosmotos movie I ever saw.** He's like a stupid man's Michael Bey – with all the awfulness that accolade promises. There's no point to the conspiracy, no burning issues that urge the conspirators on. We are left to ourselves to connect the dots from one bit of graffiti on various walls in the film to the next. Thus, the current budget crisis, the war in Iraq, Islamic extremism, the fate of social security, 47 million Americans without health care, stagnating wages, and the death of the middle class are all subsumed by the sheer terror of graffiti. A truly, stunningly idiotic film.

Graphics is far from the best part of the game. **This is the number one best TH game in the series.** Next to Underground. **It deserves strong love. It is an insane game.** There are massive levels, massive unlockable characters... it's just a massive game. **Waste your money on this game. This is the kind of money that is wasted properly.** And even though graphics suck, thats doesn't make a game good. Actually, the graphics were good at the time. Today the graphics are crap. WHO CARES? As they say in Canada, This is the fun game, aye. (You get to go to Canada in THPS3) Well, I don't know if they say that, but they might. who knows. Well, Canadian people do. Wait a minute, I'm getting off topic. This game rocks. Buy it, play it, enjoy it, love it. It's PURE BRILLIANCE.

The first was good and original. I was a not bad horror/comedy movie. So I heard a second one was made and I had to watch it . What really makes this movie work is Judd Nelson's character and the sometimes clever script. **A pretty good script for a person who wrote the Final Destination films and the direction was okay.** Sometimes there's scenes where it looks like it was filmed using a home video camera with a grainy - look. Great made - for - TV movie. **It was worth the rental and probably worth buying just to get that nice eerie feeling and watch Judd Nelson's Stanley doing what he does best.** I suggest newcomers to watch the first one before watching the sequel, just so you'll have an idea what Stanley is like and get a little history background.

[Denil et al. 2014]

Whale recognition, Kaggle Challenge

Mnih and Hinton, 2010

Image Captioning

[Vinyals et al., 2015]

reddit.com/r/deepdream

Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition
[Cadieu et al., 2014]

Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition [Cadieu et al., 2014]

CONVNETS
HOW DO THEY WORK