CS 559 Deep Learning

SGD variants, dropout regularization

Gokberk Cinbis
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

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

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

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]
Data Preprocessing
“Xavier initialization”  
[Glorot et al., 2010]

Reasonable initialization.  
(Mathematical derivation assumes linear activations)

Weight Initialization
Batch Normalization

Normalize:

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

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

\[ y(k) = \gamma(k) \hat{x}(k) + \beta(k) \]

- 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,
                                 learning_rate=1e-6, verbose=True)
```

Loss barely changing: Learning rate is probably too low
TODO

- Parameter update schemes
- Learning rate schedules
- Dropout
Parameter Updates
Training a neural network, main loop:

```python
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:

```python
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:

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

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

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

# Momentum update
v = mu * v - learning_rate * dx  # integrate velocity
x += v  # integrate position
```
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 (yet consistent) directions
- Velocity becomes damped in steep (inconsistent) direction due to quickly changing sign

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
SGD

VS

Momentum

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

Ordinary momentum update:

\[ v = \mu u * v - \text{learning\_rate} * \Delta x \] 
\[ x += v \]

Code:

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

Momentum update

- Momentum step
- Actual step
- Gradient step

Nesterov momentum update

- Momentum step
- Actual step
- "Lookahead" gradient step (bit different than original)
Nesterov Momentum update

Momentum update

Nesterov momentum update

Nesterov: the only difference...

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

\[ \theta_t = \theta_{t-1} + v_t \]
Nesterov Momentum update

\[ \nu_t = \mu \nu_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu \nu_{t-1}) \]

\[ \theta_t = \theta_{t-1} + \nu_t \]

Slightly inconvenient…
usually we have :

\[ \theta_{t-1}, \nabla f(\theta_{t-1}) \]
Nesterov Momentum update

\[
\nu_t = \mu \nu_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu \nu_{t-1})
\]

\[
\theta_t = \theta_{t-1} + \nu_t
\]

Slightly inconvenient… usually we have:
\[
\theta_{t-1}, \nabla f(\theta_{t-1})
\]

Variable transform and rearranging saves the day:

\[
\phi_{t-1} = \theta_{t-1} + \mu \nu_{t-1}
\]

Replace all thetas with phis, rearrange and obtain:

\[
\nu_t = \mu \nu_{t-1} - \epsilon \nabla f(\phi_{t-1})
\]

\[
\phi_t = \phi_{t-1} - \mu \nu_{t-1} + (1 + \mu) \nu_t
\]

# Nesterov momentum update rewrite

```python
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```
nag = Nesterov Accelerated Gradient
AdaGrad update

```python
# 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

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

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

AdaGrad update

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

[RMSProp update]

```python
# Adagrad update

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

[RMSProp]

```python
# 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
  \[ \text{MeanSquare}(w, t) = 0.9 \times \text{MeanSquare}(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w} (t) \right)^2 \]
- Dividing the gradient by \( \sqrt{\text{MeanSquare}(w, t)} \) makes the learning work much better (Tijmen Tieleman, unpublished).
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
  \[ MeanSquare(w, t) = 0.9 \times MeanSquare(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w}(t) \right)^2 \]
- Dividing the gradient by \(\sqrt{MeanSquare(w, t)}\) makes the learning work much better (Tijmen Tieleman, unpublished).

Cited by several papers as:

adagrad
rmsprop
Adam update

(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)
```

[Kingma and Ba, 2014]
Adam update
(incomplete, but close)

Looks a bit like RMSProp with momentum

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Adam update

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

---

The bias correction for Adam update is given by:

\[
\text{mb} = m / (1 - \text{beta1}^t) \quad \text{and} \quad \text{vb} = v / (1 - \text{beta2}^t)
\]

These corrections are applied to update the gradients.

---

The code snippet for Adam update is as follows:

```python
# 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)
```
Nadam update

- Adam is like RMSprop + momentum
- Nadam: replace momentum by Nesterov momentum
- Example implementation: keras.optimizers.Nadam in keras

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:**
  \[ \alpha = \frac{\alpha_0}{1 + kt} \]
Second order optimization methods

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 methods

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:

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

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

Q2: why is this impractical for training Deep Neural Nets?
Second order optimization methods

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

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

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

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

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting 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
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:

- 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:

```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
```
Regularization (dropout)
Regularization: **Dropout**

“randomly set some neurons to zero in the forward pass”

[Srivastava et al., 2014]
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)
Waaaait a second…
How could this possibly be a good idea?
Waaait a second…
How could this possibly be a good idea?

Forces the network to have a redundant representation.

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

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Waaaait a second…
How could 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, gets trained on only ~one datapoint.
At test time….

**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).

(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: \( a = w_0 x + w_1 y \)
during train:
\[
E[a] = \frac{1}{4} \left( w_0 \cdot 0 + w_1 \cdot 0 + w_0 \cdot 0 + w_1 \cdot y + w_0 \cdot x + w_1 \cdot 0 + w_0 \cdot x + w_1 \cdot y \right) \\
= \frac{1}{4} \left( 2w_0x + 2w_1y \right) \\
= \frac{1}{2} \left( w_0x + w_1y \right)
At test time....
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

during test: \( a = w_0 x + w_1 y \)
during train:
\[
E[a] = \frac{1}{4} \left( w_0 \cdot 0 + w_1 \cdot 0 \right)
+ \frac{1}{4} \left( w_0 \cdot 0 + w_1 \cdot y \right)
+ \frac{1}{4} \left( w_0 \cdot x + w_1 \cdot 0 \right)
+ \frac{1}{4} \left( w_0 \cdot x + w_1 \cdot y \right)
\]
\[
= \frac{1}{4} \left( 2 w_0 \cdot x + 2 w_1 \cdot y \right)
= \frac{1}{2} \left( w_0 \cdot x + w_1 \cdot y \right)
\]

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 \( \frac{1}{2} \)
We can do something approximate analytically

```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

=>$\text{We must scale the activations so that for each neuron:}
\text{output at test time} = \text{expected output at training time}$
Vanilla Dropout: Not recommended implementation (see notes below)

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

```python
def train_step(X):
    
    # 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
More common: “Inverted dropout”

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

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

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

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

Note: test time is unchanged!

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
More common: “Inverted dropout”

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

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
```

- Not equivalent to dropout, as $1/p$ alters the scale of the gradient
- In practice, that can be ignored when using a method that estimates per-dimension learning rate

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Next: ConvNets!