CS 559 Deep Learning

Basics of Neural Network Training

Gokberk Cinbis
Recap: activation functions

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid
Overview

1. **One time setup**
   - activation functions, preprocessing, weight initialization, regularization, gradient checking

2. **Training dynamics**
   - babysitting the learning process, parameter updates, hyperparameter optimization

3. **Evaluation**
   - model ensembles
Data Preprocessing
Step 1: Preprocess the data

(Assume $X [N \times D]$ is data matrix, each example in a row)

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data

![Graphs showing original data, decorrelated data, and whitened data](image)

- **Original data**
- **Decorrelated data** (data has diagonal covariance matrix)
- **Whitened data** (covariance matrix is the identity matrix)
In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening
Weight Initialization
- Q: what happens when W=0 init is used?
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[
W = 0.01 \times \text{np.random.randn}(D,H)
\]
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(D, H) \]

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.
Let's look at some activation statistics.

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213681
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002137
hidden layer 5 had mean 0.000002 and std 0.000552
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

*1.0 instead of *0.01
"Xavier initialization"
[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)
but when using the ReLU nonlinearity it breaks.
He et al., 2015
(note additional /2)
He et al., 2015
(note additional /2)
Proper initialization is an active area of research…

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015

…
Batch Normalization

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

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

this is a vanilla differentiable function...
Batch Normalization

“you want unit gaussian activations? just make them so.”

1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

[ioffe and Szegedy, 2015]
Batch Normalization

Usually inserted after Fully Connected / (or Convolutional, as we’ll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

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

 Normalize:

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

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

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

Note, the network can learn:

\[
\gamma(k) = \sqrt{\text{Var}[x(k)]}
\]

\[
\beta(k) = \mathbb{E}[x(k)]
\]

to recover the identity mapping.

[ioffe and Szegedy, 2015]
Batch Normalization

- 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

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

\[
\begin{align*}
\mu_\mathcal{B} &\leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean} \\
\sigma^2_\mathcal{B} &\leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \quad \text{// mini-batch variance} \\
\hat{x}_i &\leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} \quad \text{// normalize} \\
y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\end{align*}
\]
Batch Normalization

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Babysitting the Learning Process
Step 1: Preprocess the data

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

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Step 2: Choose the architecture:
say we start with one hidden layer of 50 neurons:

50 hidden neurons

CIFAR-10 images, 3072 numbers

input layer

hidden layer

output layer

10 output neurons, one per class
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train)  # disable regularization
print loss
```

loss ~2.3. “correct” for 10 classes. c.f. -log(1/10)=2.30

returns the loss and the gradient for all parameters
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, le3)
print loss
```

Loss (=objective here) went up, good. (sanity check)
Let's try to train now...

**Tip:** Make sure that you can overfit very small portion of the training data

The above code:
- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'
Lets try to train now…

**Tip**: Make sure that you can overfit very small portion of the training data.

Very small loss, train accuracy 1.00, nice!
Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

```python
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 2.392576. train: 0.88000. Val 0.10300. lr 1.000000e-06
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

```python
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 2.302502, 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 2.302496, 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: 0.266000, 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
Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

Loss barely changing: Learning rate is probably too low
Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what’s up with that? (remember this is softmax)
Let's try to train now...

I like to start with small regularization and find a learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

```python
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=1e6, verbose=True)
```

Okay now let's try learning rate 1e6. What could possibly go wrong?
Let's try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

**Loss not going down:**
learning rate too low

**Loss exploding:**
learning rate too high

Cost: NaN almost always means high learning rate...
Lets try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

loss exploding: learning rate too high

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]
Hyperparameter Optimization
Cross-validation strategy

I like to do coarse -> fine cross-validation in stages

**First stage**: only a few epochs to get rough idea of what params work

**Second stage**: longer running time, finer search

... (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever $> 3 \times$ original cost, break out early
For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
        model, two_layer_net,
        num_epochs=5, reg=reg,
        update='momentum', learning_rate_decay=0.9,
        sample_batches = True, batch_size = 100,
        learning_rate=lr, verbose=False)

    val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
    val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
    val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
    val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
    val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
    val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
    val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
    val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
    val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
    val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
    val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

Note it’s best to optimize in log space!
Now run finer search...

```python
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279464e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.868827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.026377e-04, reg: 1.228193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.493000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328310e-01, (6 / 100)
val_acc: 0.522000, lr: 5.386261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.806183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.018889e-04, (9 / 100)
val_acc: 0.495000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287607e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905640e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.633781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921764e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850665e-03, (18 / 100)
val_acc: 0.509000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.
Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.668827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.026377e-04, reg: 1.228193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.493000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.386261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.496000, lr: 1.979168e-04, reg: 1.018889e-04, (9 / 100)
val_acc: 0.496000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905640e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562008e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921764e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850665e-03, (18 / 100)
val_acc: 0.509000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?
Random Search vs. Grid Search

Grid Layout

Random Layout

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function
Monitor and visualize the loss curve

![Loss curve diagram]

- **very high learning rate**
- **low learning rate**
- **high learning rate**
- **good learning rate**

*Slide by Fei-Fei Li, Andrej Karpathy & Justin Johnson*
Loss vs. time
Loss

Bad initialization
a prime suspect
lossfunctions.tumblr.com  Loss function specimen
lossfunctions.tumblr.com
lossfunctions.tumblr.com
Monitor and visualize the accuracy:

big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?
Track the ratio of weight updates / weight magnitudes:

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

ratio between the values and updates: $\sim 0.0002 / 0.02 = 0.01$ (about okay)

want this to be somewhere around 0.001 or so
Summary

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization (random sample hyperparams, in log space when appropriate)
TODO

Look at:

- Parameter update schemes
- Learning rate schedules
- Gradient Checking
- Regularization (Dropout etc)
- Evaluation (Ensembles etc)