Lecture 38: Training Neural Networks

CS 4670/5670
Sean Bell

(Note: I'm not sure whether or not Tay.ai used neural nets)
(Recap) How do you actually train these things?

Roughly speaking:

Gather labeled data

Find a ConvNet architecture

Minimize the loss
(Recap) Training a convolutional neural network

• Split and preprocess your data
• Choose your network architecture
• Initialize the weights
• Find a learning rate and regularization strength
• Minimize the loss and monitor progress
• Fiddle with knobs
(Recap) (1) Data preprocessing

Subtract the mean

And randomly take a 224x224 sub-crop (dynamically)

*Figure: Alex Krizhevsky*
(2) Choose your architecture

Toy example: one hidden layer of size 50

CIFAR-10 images, 3072 numbers

Slide: Andrej Karpathy
(3) Initialize your weights

Set the weights to small random numbers:

\[ W = \text{np.random.randn}(D, H) \times 0.001 \]

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

\[ b = \text{np.zeros}(H) \]
(3) Check that the loss is reasonable

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

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, 0.0)  # disable regularization
print loss

returns the loss and the gradient for all parameters
(3) Check that the loss is reasonable

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

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, 1e3)  # crank up regularization
print loss

loss went up, good. (sanity check)
(4) Overfit a small portion of the data

Details:

'sgd': vanilla gradient descent (no momentum etc)

learning_rate_decay = 1: constant learning rate

sample_batches = False (full gradient descent, no batches)

ePOCHs = 200: number of passes through the data
(4) Overfit a small portion of the data

100% accuracy on the training set (good)
(4) Find a learning rate

Let’s start with small regularization and find the learning rate that makes the loss decrease:

```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)
```
(4) Find a learning rate

```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)
```
(4) Find a learning rate

Loss barely changes  Why is the accuracy 20%?
(learning rate is too low or regularization too high)

Slide: Andrej Karpathy
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

Loss is NaN $\rightarrow$ learning rate is too high
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: 3e-3

Loss is inf —> still too high
But now we know we should be searching the range [1e-5 ... 1e-3]
(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search

Tip: if loss > 3 * original loss, quit early (learning rate too high)
(4) Find a learning rate

Coarse to fine search
(4) Find a learning rate

Coarse to fine 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: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
    val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
    val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
    val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
    val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
    val_acc: 0.460000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
    val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
    val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
    val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
    val_acc: 0.490000, 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: 3.905040e-02, (12 / 100)
    val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562080e-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: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
    val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
    val_acc: 0.500000, 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)
```

Remember this is just a 2 layer neural net with 50 neurons.

- 53%
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

(Why linearly?)

Larger learning rates tend to look more exponential

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batchsize — if too noisy, increase the batch size

Possibly too linear (learning rate too small)

Figure: Andrej Karpathy
(4) Find a learning rate

Figure: Andrej Karpathy
(4) Find a learning rate

Figure: Andrej Karpathy
lossfunctions

They are a window to your model’s heart.

Contribute loss functions to @karpathy. It doesn’t matter if your loss functions are flat, converge, diverge, step or oscillate (or any combination of the above). All loss functions are computed beautiful in their own way and are sought after with equal tenacity.

ARCHIVE
Ah, the Sharp Corner Loss (SCL). Bad initialization a prime suspect.
A beautiful rainbow of learning! This code is definitely bug free. Learning rate decay might be slightly too high.
Looks good but mysterious high frequency harmonics are mysterious
When you stare at loss functions for a while you start seeing things
(4) Find a learning rate

Visualize the accuracy

**Big gap:** overfitting
(increase regularization)

**No gap:** underfitting
(increase model capacity, make layers bigger or decrease regularization)

*Figure: Andrej Karpathy*
(4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the weights

Nice clean weights: training is proceeding well

*Figure: Alex Krizhevsky, Andrej Karpathy*
Learning rate schedule

How do we change the learning rate over time?

Various choices:

• Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])

• Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])

• Scale by \( \sqrt{1-t/\text{max}_t} \) (used by BVLC to re-implement GoogLeNet)

• Scale by \( 1/t \)

• Scale by \( \exp(-t) \)
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization
Questions?
Tricks for making training work better
Momentum

**Simple but powerful improvement:** Give some “momentum” to the parameters

\[
v_{i+1} = 0.9v_i - \alpha \frac{\partial L}{\partial \theta}(\theta_i)
\]

\[
\theta_{i+1} = \theta_i + v_{i+1}
\]

**Unfortunate nomenclature:** the damping factor is called “momentum”

**“Lesson from the trenches”:** well-tuned SGD with Momentum is very hard to beat for ConvNets

*Figure: Andrej Karpathy*
Momentum

Intuition behind momentum:

- Imagine a ball on the loss surface (its position is the current weight settings)

- Directions with lots of oscillations are damped

- Builds up speed in directions with a consistent gradient

Figure: Andrej Karpathy
“RMSprop”

On Geoff Hinton’s coursera lecture 6a, he mentioned various “tricks” including “rmsprop”

**Idea:** track the moving average of squared gradients

```python
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / np.sqrt(cache + 1e-8)
```

decay_rate is a hyper-parameter (typically 0.9, 0.99, or 0.999)