Administrative

- **A2** is out. It was late 2 days so due date will be shifted by ~2 days.

- we updated the project page with many pointers to datasets.
Lecture 6:

Training Neural Networks
impulses carried toward cell body

dendrites

branches of axon

axon terminals

nucleus

cell body

impulses carried away from cell body

$x_0$

$w_0$

axon from a neuron

synapse

$w_0 x_0$

dendrite

$w_1 x_1$

$w_2 x_2$

$\sum_i w_i x_i + b$

$\mathbf{f}(\sum_i w_i x_i + b)$

output axon

activation function
Backpropagation
(recursive chain rule)
Mini-batch Gradient descent

Loop:
1. Sample a batch of data
2. Backprop to calculate the analytic gradient
3. Perform a parameter update
A bit of history

Widrow and Hoff, ~1960: Adaline
A bit of history

Rumelhart et al. 1986: First time back-propagation became popular

\[
E_p = \frac{1}{2} \sum_{j} (o_{pj} - a_{pj})^2
\]  

be our measure of the error on input/output pattern \( p \) and let \( E = \sum p E_p \) be our overall measure of the error. We wish to show that the delta rule implements a gradient descent \( E \) when the units are linear. We will proceed by simply showing that

\[
- \frac{\partial E_p}{\partial w_{j}} = \delta_{pj} y_j^	op,
\]

which is proportional to \( \Delta_j \) \( w_{j} \) as prescribed by the delta rule. When there are \( \alpha \) hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output \( \alpha \) of the unit times the derivative of the output with respect to the weight.

\[
\frac{\partial E_p}{\partial w_{j}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{j}}.
\]

The first part tells how the error changes with \( \alpha \) of the \( j \)th unit and the second part tells how much changing \( w_{j} \) changes that output. Now, the derivatives are easy to compute. First, from Equation 2

\[
\frac{\partial E_p}{\partial o_{pj}} = -(o_{pj} - a_{pj}) = - \delta_{pj}.
\]

Not surprisingly, the contribution of unit \( \alpha \) to the error is simply proportional to \( \delta_{pj} \). Moreover, since we have linear units,

\[
o_{pj} = \sum_{j} w_{j} y_j,
\]

from which we conclude that

\[
\frac{\partial o_{pj}}{\partial w_{j}} = y_j.
\]

Thus, substituting back into Equation 3, we see that

\[
- \frac{\partial E_p}{\partial w_{j}} = \delta_{pj} y_j^	op.
\]
A bit of history

[Hinton and Salkhutdinov 2006]

Reinvigorated research in Deep Learning
Training Neural Networks
Step 1: Preprocess the data

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

- **Original data**
- **Zero-centered data**
  
- **Normalized data**

$$X = \text{np.mean}(X, \text{axis} = 0)$$

$$X /\!\!/ = \text{np.std}(X, \text{axis} = 0)$$
Step 1: Preprocess the data

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

- Original data
- Decorrelated data: (data has diagonal covariance matrix)
- Whitened data: (covariance matrix is the identity matrix)
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
Before we try training, let's **initialize** well:

- set weights to small random numbers

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

  (Matrix of small numbers drawn randomly from a gaussian)

  **Warning:** *This is not optimal, but simplest! (More on this later)*

- set biases to zero
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
2.30261216167
```

loss ~2.3. “correct” for 10 classes

returns the loss and the gradient for all parameters
Double 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)  # crank up regularization
print loss

3.06859716482

loss went up, good. (sanity check)
Let's try to train now…

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

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

**Details:**
- (learning_rate_decay = 1 means no decay, the learning rate will stay constant)
- sample_batches = False means we’re doing full gradient descent, not mini-batch SGD
- we’ll perform 200 updates (epochs = 200)

“Epoch”: number of times we see the training set
Lets try to train now…

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

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

**loss exploding:**
learning rate too high

```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)
```
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

Loss barely changing: Learning rate must be too low. (could also be reg too high)

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 learning rate that makes the loss go down.

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

**loss exploding:**
learning rate too high

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…
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

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

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]
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 * original cost, break out early
For example: run coarse search for 5 epochs

```python
max_count = 100
for count in xrange(max_count):
    reg = 100**uniform(-5, 5)
    lr = 100**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: 5.401607e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```
Now run finer search...

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: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.228193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.490000, 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: 5.588183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.618888e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.466271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.87807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.906540e-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: 3.148888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.436349e-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: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.460000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.510000, 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...

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

But this best cross-validation result is worrying. Why?
Normally you can’t afford a huge computational budget for expensive cross-validations. Need to rely more on intuitions and visualizations…

**Visualizations to play with:**
- **loss** function
- validation and training **accuracy**
- min,max,std for **values and updates**, (and monitor their ratio)
- **first-layer visualization** of weights (if working with images)
Monitor and visualize the loss curve

If this looks too linear: learning rate is low.
If it doesn’t decrease much: learning rate might be too high
Monitor and visualize the loss curve

If this looks too linear: learning rate is low.
If it doesn’t decrease much: learning rate might be too high

the “width” of the curve is related to the batch size. This one looks too wide (noisy)
=> might want to increase batch size
Monitor and visualize the accuracy:

- **big gap** = overfitting
  - => increase regularization strength

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

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.01 - 0.001 or so
Visualizing first-layer weights:

Noisy weights =>
Regularization maybe not strong enough
(Regarding your Assignment #1)

=> Regularization not strong enough
So far:

- We’ve seen the process for performing cross-validations

- There are several things we can track to get intuitions about how to do it more efficiently
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/L1/Maxnorm/Dropout)
- loss to use (e.g. SVM/Softmax)
- initialization

neural networks practitioner
music = loss function
Initialization
- becomes more tricky and important in deeper networks. Usually approx. \( W \sim N(0, 0.01) \) works. If not:

Consider what happens to the output distribution of neurons with different number of inputs (low or high)
Initialization

- becomes more tricky and important in deeper networks. Usually approx. $W \sim N(0, 0.01)$ works. If not:

$$w = \text{np.random.randn}(n) \times \sqrt{2.0/n}$$

normalize by square root of the number of incoming connections (fan in)

=> ensures equal variance of each neuron in network

(tricky, subtle, but very important topic. See notes for details)
Regularization knobs
- L2 regularization
- L1 regularization
- L1 + L2 can also be combined
- Max norm constraint

\[ \frac{1}{2} \lambda w^2 \]
\[ \lambda |w| \]

L1 is “sparsity inducing” (many weights become almost exactly zero)

enforce maximum L2 norm of the incoming weights
Seemingly unrelated: **Model Ensembles**
- One way to *always* improve final accuracy: take several trained models and average their predictions
Regularization: **Dropout**

“randomly set some neurons to zero”

(a) Standard Neural Net

(b) After applying dropout.

[Srivastava et al.]
\[
p = 0.5 \quad \text{# probability of keeping a unit active. higher = less dropout}
\]

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)
At test time we don’t drop, but have to be careful:

```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

if the output of a neuron is $x$ but the probability of keeping it is only $p$, then the output of the neuron (in expectation) is:

$$px + (1-p)0 = px$$
More common: “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
Learning rates and updates:
- SGD + Momentum > SGD
- Momentum 0.9 usually works well
- **Decrease the learning rate over time** (people use $1/t$, $\exp(-t)$, or steps)

simplest: $\text{learning\_rate} *= 0.97$ every epoch (or so)
Summary

- Properly preprocess the data
- Run cross-validations across many tips/tricks
- Use visualizations to guide the ranges and cross-val
- Ensemble multiple models and report test error
Next Lecture:

Convolutional Neural Networks