Getting Neural Nets to Train
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley
This lecture

Help! My network doesn’t train

If you follow everything I described in the previous lectures...
And you implement everything correctly...
And you train everything for a long time...

There is a good chance it still won’t work
Neural networks are messy
They require lots of “tricks” to train well
We’ll discuss these tricks today

- Normalizing inputs and outputs
- Normalizing activations (batch normalization)
- Initialization of weight matrices & bias vectors
- Gradient clipping
- Best practices for hyperparameter optimization
- Ensembling, dropout
The dangers of **big** inputs, activations, and outputs

Why?
The dangers of **big** inputs, activations, and outputs

\[
\frac{dL}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{dL}{dz^{(1)}} = \delta x^T
\]

Gradient will get scaled by big \(x\) values

"easy" case

\[
W^{(1)} b^{(1)}
\]

\[
x \rightarrow \text{linear layer} \rightarrow z^{(1)}
\]

ReLU

\[
a^{(1)} \rightarrow \text{linear layer} \rightarrow z^{(2)}
\]

softmax

\[\mathcal{L}(z^{(2)})\]

"hard" case

\[
W^{(2)} b^{(2)}
\]

\[
W^{(1)} b^{(1)}
\]
The dangers of big inputs, activations, and outputs

"easy" case

\[
\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{d\mathcal{L}}{dz^{(1)}} = \delta x^T
\]

"hard" case

\[
\nabla_{\theta} \mathcal{L}(\theta_0)
\]
The dangers of big inputs, activations, and outputs

In general...

we really want all entries in $x$ to be roughly on the same scale
sometimes not a problem:

images: all pixels are roughly in $[0, 1]$ or $\{0, \ldots, 255\}$
discrete inputs (e.g., NLP): all inputs are one-hot (zero or one)
sometimes a huge problem:

forecasting the weather?
temperature: somewhere 40-100?
humidity: somewhere 0.3 - 0.6?

etc.
The dangers of **big** inputs, activations, and outputs

What can we do?

**Standardization**: transform inputs so they have $\mu = 0, \sigma = 1$

To make $\mu = 0$: $\tilde{x}_i = x_i - E[x]$  
$E[x] \approx \frac{1}{N} \sum_{i=1}^{N} x_i$

To *also* make $\sigma = 1$: $\tilde{x}_i = \frac{x_i - E[x]}{\sqrt{E[(x_i - E[x])^2]}}$  
standard deviation

...and activations??

and outputs! (if doing regression)

all operations are per-dimension
Standardizing activations?

What can we do?

**Standardization**: transform inputs so they have \( \mu = 0, \sigma = 1 \)

What if we start getting really different scales for each dimension here?

Can we just standardize these activations too?

Basically yes, but now the mean and standard deviation changes during training...
Standardizing activations?

Basic idea:

\[ z_i^{(1)} = W^{(1)} x_i + b^{(1)} \]

\[ a_i^{(1)} = \text{ReLU}(z_i^{(1)}) \]

\[ \mu^{(1)} = \frac{1}{N} \sum_{i=1}^{N} a_i^{(1)} \]

\[ \sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^{N}(a_i^{(1)} - \mu^{(1)})^2} \]

\[ \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \]

\[ z_i^{(2)} = W^{(2)} \bar{a}_i^{(1)} + b^{(2)} \]

e etc...

This seems very expensive, since we don’t want to evaluate all points in the dataset every gradient step!
Batch normalization (basic version)

Basic idea:

\[ z_i^{(1)} = W^{(1)} x_i + b^{(1)} \]

\[ a_i^{(1)} = \text{ReLU}(z_i^{(1)}) \]

\[ \mu^{(1)} = \frac{1}{N} \sum_{i=1}^{N} a_i^{(1)} \]

\[ \sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (a_i^{(1)} - \mu^{(1)})^2} \]

\[ \tilde{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \]

\[ z_i^{(2)} = W^{(2)} \tilde{a}_i^{(1)} + b^{(2)} \]

This seems very expensive, since we don’t want to evaluate all points in the dataset every gradient step. Compute mean and std only over the current batch.

etc...
Batch normalization (real version)

Basic idea:

\[ z_i^{(1)} = W^{(1)} x_i + b^{(1)} \]

\[ a_i^{(1)} = \text{ReLU}(z_i^{(1)}) \]

\[ \mu^{(1)} = \frac{1}{N} \sum_{i=1}^{N} a_i^{(1)} \]

\[ \sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (a_i^{(1)} - \mu^{(1)})^2} \]

\[ \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \]

\[ z_i^{(2)} = W^{(2)} \bar{a}_i^{(1)} + b^{(2)} \]

eetc...

This seems very expensive, since we don’t want to evaluate all points in the dataset every gradient step.

\[ \mu^{(1)} \approx \frac{1}{B} \sum_{j=1}^{B} a_{ij}^{(1)} \]

\[ \sigma^{(1)} \approx \sqrt{\frac{1}{B} \sum_{j=1}^{B} (a_{ij}^{(1)} - \mu^{(1)})^2} \]

\[ \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \gamma + \beta \]

learnable scale and bias
same dim as \( \bar{a}_i^{(1)} \)
Batch normalization “layer”

$\mu^{(1)} \approx \frac{1}{B} \sum_{j=1}^{B} a_{i,j}^{(1)}$

$\sigma^{(1)} \approx \sqrt{\frac{1}{B} \sum_{j=1}^{B} (a_{i,j}^{(1)} - \mu^{(1)})^2}$

$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \gamma + \beta$

How to train?

Just use backpropagation!

**Exercise:** figure out the derivatives w.r.t. parameters and input!
Where to put back normalization?
Where to put back normalization?

- Scale and bias seemingly should be subsumed by the next linear layer?
- All ReLU outputs are positive
- The “classic” version
- Just appears to be a transformation on the preceding linear layer?

No one seems to agree on what the right way to do it is, try a few options and see what works (but both often work)
A few considerations about batch norm

Often we can use a larger learning rate with batch norm.

Models with batch norm can train much faster.

Generally requires less regularization (e.g., doesn’t need dropout).

Very good idea in many cases.
Weight initialization
We want the overall scale of activations in the network not to be too big or too small for our initial (randomized) weights, so that the gradients propagate well.

**Basic initialization methods:** ensure that activations are on a reasonable scale, and the scale of activations doesn’t grow or shrink in later layers as we increase the number of layers.

**More advanced initialization methods:** try to do something about eigenvalues of Jacobians.

\[
\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \cdot \frac{da^{(1)}}{dz^{(1)}} \cdot \frac{dz^{(2)}}{da^{(1)}} \cdot \frac{d\mathcal{L}}{dz^{(2)}}
\]

\[
\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \ldots J_n \frac{d\mathcal{L}}{dz^{(n)}}
\]

*If we multiply many many numbers together, what will we get?*

- If most of the numbers are < 1, we get 0
- If most of the numbers are > 1, we get infinity

We only get a reasonable answer if the numbers are all close to 1!
Basic initialization

Simple choice: Gaussian random weights

\[ W^{(i)}_{jk} \sim \mathcal{N}(0, 0.0001) \]

Ideally we could just initialize here
But we have no idea where that is!

Goal is **not** to start at a good solution, but to have well-behaved gradients & activations

Why is this bad?

\[
\frac{d\mathcal{L}}{dW^{(i)}} = \frac{dz^{(i)}}{dW^{(i)}} \frac{d\mathcal{L}}{dz^{(i)}} = \delta^{(i-1)^T}
\]
Basic initialization

\[ W_{ij} \sim \mathcal{N}(0, \sigma_W^2) \]
\[ b_i \approx 0 \]

reasonable choice!  
if we standardize \( x \), then  
\[ x \sim \mathcal{N}(0, 1) \]

what is (roughly) the magnitude of \( z_i \)?

\[ z_i = \sum_j W_{ij}a_j + b_i \approx 0 \]

assume \( a_j \sim \mathcal{N}(0, \sigma_a) \)

everything is (roughly) 0-mean

\[ E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2 \]

dimensionality of \( a \)

if \( D_a \sigma_W^2 > 1 \), magnitude grows with each layer!
if \( D_a \sigma_W^2 < 1 \), magnitude shrinks with each layer!

what if we choose \( \sigma_W^2 = 1/D_a \)?
Basic initialization

\[ E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2 \]

dimensionality of \( a \)

- if \( D_a \sigma_W^2 > 1 \), magnitude grows with each layer!
- if \( D_a \sigma_W^2 < 1 \), magnitude shrinks with each layer!
- what if we choose \( \sigma_W^2 = 1/D_a \)?

basic principle: get std of \( W_{ij} \) to be about \( 1/\sqrt{D_a} \) this sometimes referred to as “Xavier initialization”
Little detail: ReLUs

\[ E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2 \]

basic principle: get std of \( W_{ij} \) to be about \( 1/\sqrt{D_a} \)

This was all without nonlinearities!

problem: \( a_j = \text{ReLU}(z_j) \)

“negative half” of 0-mean activations is removed!
variance is cut in half!
might not seem like much...
but it adds up!

Image from: Fei-Fei Li & Andrej Karpathy
Little detail: ReLUs

\[ E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2 \]

basic principle: get std of \( W_{ij} \) to be about \( \frac{1}{\sqrt{D_a}} \)

\[ \frac{1}{\sqrt{\frac{1}{2} D_a}} \]

This was all without nonlinearities!

\[ a_j = \text{ReLU}(z_j) \]

“negative half” of 0-mean activations is removed! variance is cut in half!

might not seem like much...

proposed by He et al. for ResNet makes big difference 150+ layers...

Image from: Fei-Fei Li & Andrej Karpathy
Littler detail: ReLUs & biases

$$W_{ij} \sim \mathcal{N}(0, \sigma_W^2)$$

Problem: $a_j = \text{ReLU}(z_j)$

half of our units (on average) will be “dead”!

often initialize $b_i = 0.1$ (or small constant)
Advanced initialization

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1, we get 0
If most of the numbers are > 1, we get infinity
We only get a reasonable answer if the numbers are all close to 1!

for each $J_i$, we can write: $J_i = U_i \Lambda_i V_i$

e.g., using singular value decomposition

diagonal matrix with same eigenvalues as $J_i$

scale-preserving transformations (i.e., orthonormal bases)
Advanced initialization

If we multiply many many numbers together, what will we get?

- If most of the numbers are < 1, we get 0
- If most of the numbers are > 1, we get infinity
- We only get a reasonable answer if the numbers are all close to 1!

\[
\frac{dL}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{dL}{dz^{(2)}}
\]

\[
\frac{dL}{dW^{(1)}} = J_1 J_2 J_3 \ldots J_n \frac{dL}{dz^{(n)}}
\]

for each \( W^{(i)} \), we can write:

\[
W^{(i)} = U^{(i)} \Lambda^{(i)} V^{(i)}
\]

e.g., using singular value decomposition

\[
W^{(i)} \leftarrow U^{(i)} V^{(i)}
\]

just need to force this to be identity matrix

even simpler:

```python
# arbitrary random matrix (doesn't really matter how)
a = get_rng().normal(0.0, 1.0, flat_shape)
u, _, v = np.linalg.svd(a, full_matrices=False)
# pick the one with the correct shape
q = u if u.shape == flat_shape else v
```

needed if non-square

guaranteed orthonormal

https://github.com/Lasagne/Lasagne
Last bit: Gradient clipping

- Took a step that was too big in the wrong place
- Something got divided by something small (e.g., in batch norm, softmax, etc.)
- Just got really unlucky
Clipping the monster gradients

per-element clipping:  \[ \bar{g}_i \leftarrow \max(\min(g_i, c_i), -c_i) \]

norm clipping:  \[ \bar{g}_i \leftarrow g \frac{\min(||g||, c)}{||g||} \]

how to choose \( c \)?

run a few epochs (assuming it doesn’t explode)
see what “healthy” magnitudes look like
Ensembles & dropout
What if my model makes a mistake?

**Problem:** neural networks have many parameters, often have high variance

Not nearly as high as we would expect from basic learning theory (i.e., overfitting is usually **not** catastrophic), but still...

**Interesting idea:** when we have multiple high-variance learners, maybe they’ll **agree** on the right answer, but **disagree** on the wrong answer

**Said another way:** there are many more ways to be wrong than to be right
Ensembles in theory

\[
\text{Variance} = E_{D \sim p(D)}[| |f_D(x) - \bar{f}(x)||^2]
\]

\[
\bar{f}(x) = E_{D \sim p(D)}[f_D(x)] \approx \frac{1}{M} \sum_{i=1}^{M} f_{D_j}(x)
\]

can we actually estimate this thing?  where do we get \( M \) different datasets??

Can we \textbf{cook up} multiple independent datasets from a single one?

**Simple approach:** just chop a big dataset into \( M \) non-overlapping parts

\[
D = \{(x_i, y_i)\}
\]

for each \( D_j \) pick \( N \) indices randomly in \{1, ..., N\} \( i_{j,1}, \ldots, i_{j,N} \)

\[
D_j = \{(x_{i_{j,1},1}, y_{i_{j,1}}), (x_{i_{j,2},2}, y_{i_{j,2}}), \ldots, (x_{i_{j,N},N}, y_{i_{j,N}})\}
\]

overlapping but \textbf{independently sampled}

turns out we actually don’t need this!
Ensembles in theory

\[ \mathcal{D} = \{(x_i, y_i)\} \]

for each \( \mathcal{D}_j \) pick \( N \) indices randomly in \( \{1, ..., N\} \) \( i_{j,1}, ... i_{j,N} \)

\[ \mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}})(x_{i_{j,2}}, y_{i_{j,2}}), ..., (x_{i_{j,N}}, y_{i_{j,N}})\} \]

This is called resampling \textbf{with replacement}

\[ \mathcal{D} \]
\[ x_1 \ x_2 \ x_3 \]

\[ \mathcal{D}_1 \]
\[ x_2 \ x_3 \ x_3 \]  
\[ \text{\begin{tabular}{c}
\ 2 \\
\ 3 \\
\ 3 
\end{tabular}} \]

\[ \mathcal{D}_2 \]
\[ x_3 \ x_1 \ x_3 \]  
\[ \text{\begin{tabular}{c}
\ 3 \\
\ 1 \\
\ 3 
\end{tabular}} \]

train separate models on each \( \mathcal{D}_j \)
Ensembles in theory

\[ \mathcal{D} = \{(x_i, y_i)\} \]

for each \( \mathcal{D}_j \) pick \( N \) indices randomly in \( \{1, ..., N\} \ i_{j,1}, ... i_{j,N} \)

\[ \mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}}), (x_{i_{j,2}}, y_{i_{j,2}}), ..., (x_{i_{j,N}}, y_{i_{j,N}})\} \]

train separate models on each \( \mathcal{D}_j \)

\[ p_{\theta_1}(y|x), ..., p_{\theta_M}(y|x) \]

how do we predict?

**principled** approach: average the probabilities:

\[ p(y|x) = \frac{1}{M} \sum_{j=1}^{M} p_{\theta_j}(y|x) \]

**simple** approach: majority vote
Ensembles in practice

There is already a lot of randomness in neural network training

- Random initialization
- Random minibatch shuffling
- Stochastic gradient descent

In practice we get much of the same benefit without resampling

Train $M$ models $p_{\theta_j}(y|x)$ on the same $\mathcal{D}$

$$p(y|x) = \frac{1}{M} \sum_{j=1}^{M} p_{\theta_j}(y|x) \quad \text{or majority vote}$$
Even **faster** ensembles

features (somewhat task-agnostic)
often the most expensive part

task-specific classification layers

share the features for all models in the ensemble

separate ensemble of classifier “heads”
Even faster ensembles:

snapshot ensembles:

save out parameter snapshots over the course of SGD optimization, use each snapshot as a model in the ensemble

**advantage:** don’t need to have a bunch of separate training runs

...but need to set things up carefully so that the snapshots are actually different

**combining predictions:** could average probabilities or vote, or just average the parameter vectors together

Huang et al., Snapshot Ensembles: Train 1, Get M For Free
Some comparisons

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction method</th>
<th>Test Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (10 epochs)</td>
<td>Single model</td>
<td>0.837</td>
</tr>
<tr>
<td>True ensemble of 10 models</td>
<td>Average predictions</td>
<td>0.855</td>
</tr>
<tr>
<td>True ensemble of 10 models</td>
<td>Voting</td>
<td>0.851</td>
</tr>
<tr>
<td>Snapshots (25) over 10 epochs</td>
<td>Average predictions</td>
<td>0.865</td>
</tr>
<tr>
<td>Snapshots (25) over 10 epochs</td>
<td>Voting</td>
<td>0.861</td>
</tr>
<tr>
<td>Snapshots (25) over 10 epochs</td>
<td>Parameter averaging</td>
<td>0.864</td>
</tr>
</tbody>
</table>

your mileage may vary
Really really big ensembles?

The bigger the ensemble is, the better it works (usually)

But making huge ensembles is expensive

Can we make multiple models **out of a single neural network**?

**Dropout**

randomly set some activations to zero in the forward pass

“new” network made out of the old one
Dropout

randomly set some activations to zero in the forward pass

Implementation:

for each $a_j^{(i)}$, set it to $a_j^{(i)} m_{ij}$

$m_{ij} \sim \text{Bernoulli}(0.5)$

1.0 with probability 50%, 0.0 otherwise

Andrej Karpathy
Dropout

randomly set some activations to zero in the forward pass

How could this possibly work?

Can think of every dropout mask as defining a different model

Hence this looks like a huge ensemble

How huge?
At test time...

During training:

for each $a_j^{(i)}$, set it to $a_j^{(i)} m_{ij}$

$m_{ij} \sim \text{Bernoulli}(0.5)$

At test time: want to combine all the models

could just generate many dropout masks

what if we stop dropping out at test time?

before: on average $\frac{1}{2}$ of dimensions are forced to 0

now: none of them are, so $W^{(i)} a^{(i)}$ will be $\approx 2 \times$ bigger

solution: $\tilde{W}^{(i)} = \frac{1}{2} W^{(i)}$ (divide all weights by 2!)

Andrej Karpathy
With all these tricks, we have a lot of hyperparameters

Some of these affect optimization (training)
- Learning rate
- Momentum
- Initialization
- Batch normalization

Some of these affect generalization (validation)
- Ensembling
- Dropout
- Architecture (# and size of layers)

How do we pick these?
- Recognize which is which: this can really matter!
  - Bad learning rate, momentum, initialization etc. shows up very early on in the training process
  - Effect of architecture usually only apparent after training is done
- Coarse to fine: start with broad sweep, then zero in
- Consider random hyperparameter search instead of grid

Example: short (5 epoch) log-space LR & weight decay sweep

Andrej Karpathy