Lecture #04 – Training Deep Neural Networks

Aykut Erdem // Hacettepe University // Spring 2020

visualization of mode connectivity for ResNet-20 with no skip connections on CIFAR-10 dataset, Javier Ideami
Previously on CMP784

• multi-layer perceptrons
• activation functions
• chain rule
• backpropagation algorithm
• computational graph
• distributed word representations
Lecture overview

• data preprocessing and normalization
• weight initializations
• ways to improve generalization
• optimization
• babysitting the learning process
• hyperparameter selection

Disclaimer: Much of the material and slides for this lecture were borrowed from
— Fei-Fei Li, Andrej Karpathy and Justin Johnson’s CS231n class
— Roger Grosse’s CSC321 class
— Shubhendu Trivedi and Risi Kondor’s CMSC 35246 class
— Efstratios Gavves and Max Welling’s UvA deep learning class
— Hinton’s Neural Networks for Machine Learning class
Paper presentations start next week

• Paper presentations will start next week!

• Quizzes
  – 1 or 2 short answer questions about the paper (10 mins long)

Training Deep Neural Networks
The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks
Paper presentations start next week

- Paper presentations will start next week!
- Quizzes
  - 1 or 2 short answer questions about the paper (10 mins long)
- Paper critiques

Training Deep Neural Networks The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks
Paper Critiques

• A short summary of the paper,
• Main contributions of the paper,
• Write a 4 point review

✓ Very detailed experimental section. I liked experiments showing object detection helps saliency estimation
✓ The methods seems applicable to a wide rage of problems beyond saliency estimation
✗ I had a hard time understanding section 3.1: Why do the authors estimate object saliency after already classifying the object?
✗ The mathematical notation in technical section is inconsistent. Objects are referred as x in 3.0 and y in 3.1+. 

Adapted from Philipp Krähenbühl
Activation Functions
Activation Functions

Sigmoid

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

\text{tanh} \quad \text{tanh}(x)

ReLUs

\[ \text{ReLU} = \max(0, x) \]

Leaky ReLU

\[ \max(0.1x, x) \]

Maxout

\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

ELUs

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]
Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
Activation Functions

• Squashes numbers to range \([0,1]\)
• Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:
1. Saturated neurons “kill” the gradients

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

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

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:
1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
Consider what happens when the input to a neuron \((x)\) is always positive:

What can we say about the gradients on \(w\)?
Consider what happens when the input to a neuron \((x)\) is always positive:

\[
f \left( \sum_i w_i x_i + b \right)
\]

What can we say about the gradients on \(w\)?
Always all positive or all negative :(
(this is also why you want zero-mean data!)
Activation Functions

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

- Squashes numbers to range $[0,1]$
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:
1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. $\exp()$ is a bit compute expensive
Activation Functions

- Squashes numbers to range \([-1, 1]\)
- zero centered (nice)
- still kills gradients when saturated :(

\[ \tanh(x) \]

[LeCun et al., 1991]
Activation Functions

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions

- Computes $f(x) = \max(0,x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:

**Hint:** what is the gradient when $x < 0$?

[Krizhevsky et al., 2012]
DATA CLOUD

active ReLU

dead ReLU
will never activate
→ never update
→ people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

dead ReLU will never activate → never update
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Leaky ReLU
\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)
\[ f(x) = \max(\alpha x, x) \]
backprop into \( \alpha \)
(parameter)

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Exponential Linear Units (ELU)

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires $\exp()$

$$f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha(\exp(x) - 1) & \text{if } x \leq 0 
\end{cases}$$

[Clevert et al., 2015]
Activation Functions

Scaled Exponential Linear Units (SELU)

- Scaled version of ELU
- Stable and attracting fixed points for the mean and variance
- No need for batch normalization
- ~100 pages long of pure math

Using the Banach fixed-point theorem, we prove that activations close to zero mean and unit variance that are propagated through many network layers will converge towards zero mean and unit variance — even under the presence of noise and perturbations."

\[ f(x) = \lambda \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha (\exp(x) - 1) & \text{if } x \leq 0 
\end{cases} \]

\[ \alpha = 1.6732632423543772848170429916717 \]

\[ \lambda = 1.0507009873554804934193349852946 \]

[Klambauer et al., 2017]
Data Preprocessing and Normalization
Data preprocessing

• Scale input variables to have similar diagonal covariances $c_i = \sum_j (x_{i}^{(j)})^2$
  
  – Similar covariances $\rightarrow$ more balanced rate of learning for different weights
  
  – Rescaling to 1 is a good choice, unless some dimensions are less important

\[ x = [x^1, x^2, x^3]^T, \theta = [\theta^1, \theta^2, \theta^3]^T, a = \tanh(\theta^T x) \]

\[ x^1, x^2, x^3 \rightarrow \text{much different covariances} \]

\[ \frac{\partial L}{\partial \theta} \bigg|_{x^1, x^2, x^3} : \text{much different} \]

\[ \text{Generated gradients} \]

\[ \text{Gradient update harder: } \theta^{t+1} = \theta^t - \eta_t \begin{bmatrix} \frac{\partial L}{\partial \theta^1} \\ \frac{\partial L}{\partial \theta^2} \\ \frac{\partial L}{\partial \theta^3} \end{bmatrix} \]
Data preprocessing

• Input variables should be as decorrelated as possible
  – Input variables are “more independent”
  – Network is forced to find non-trivial correlations between inputs
  – Decorrelated inputs → Better optimization
  – Obviously not the case when inputs are by definition correlated (sequences)

• Extreme case
  – Extreme correlation (linear dependency) might cause problems [CAUTION]
Data preprocessing

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

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)
Data preprocessing

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

- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
  - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to $\sqrt{\text{fan-in}}$.

- We can also scale the learning rate the same way. More on this later!

(from Hinton’s notes)
but when using the ReLU nonlinearity it breaks.
He et al., 2015
(note additional /2)
Proper initialization is an active area of research...

- Understanding the difficulty of training deep feedforward neural networks. Glorot and Bengio, 2010
- Random walk initialization for training very deep feedforward networks. Sussillo and Abbott, 2014
- Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. He et al., 2015
- Data-dependent Initializations of Convolutional Neural Networks. Krähenbühl et al., 2015
- All you need is a good init. Mishkin and Matas, 2015
- How to start training: The effect of initialization and architecture. Hanin and Rolnick, 2018
- How to Initialize your Network? Robust Initialization for WeightNorm & ResNets. Arpit et al., 2019

...
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:

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

this is a vanilla differentiable function...

[ioffe and Szegedy, 2015]
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]
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?

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

[ioffe and Szegedy, 2015]
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: $\mathcal{B} = \{x_1 \ldots m\}$; Parameters to be learned: $\gamma, \beta$

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

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

[ioffe and Szegedy, 2015]
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)

[ioffe and Szegedy, 2015]
Other normalization schemes

- **Layer Normalization**

- **Weight Normalization**

- **Instance Normalization**
  Ulyanov et al., Instance normalization: The missing ingredient for fast stylization. arXiv preprint, 2016

- **Batch Renormalization**
  Ioffe, Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models, NIPS 2017

- **Group Renormalization**
  Wu and He, Group Normalization, ECCV 2018
Improving Generalization
Preventing Overfitting

• **Approach 1:** Get more data!
  – Almost always the best bet if you have enough compute power to train on more data.

• **Approach 2:** Use a model that has the right capacity:
  – enough to fit the true regularities.
  – not enough to also fit spurious regularities (if they are weaker).

• **Approach 3:** Average many different models.
  – Use models with different forms.
  – Or train the model on different subsets of the training data (this is called “bagging”).

• **Approach 4: (Bayesian)** Use a single neural network architecture, but average the predictions made by many different weight vectors.
Some ways to limit the capacity of a neural net

• The capacity can be controlled in many ways:
  • **Architecture:** Limit the number of hidden layers and the number of units per layer.
  • **Early stopping:** Start with small weights and stop the learning before it overfits.
  • **Weight-decay:** Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
  • **Noise:** Add noise to the weights or the activities.

• Typically, a combination of several of these methods is used.
Regularization

• Neural networks typically have thousands, if not millions of parameters
  – Usually, the dataset size smaller than the number of parameters
• Overfitting is a grave danger
• Proper weight regularization is crucial to avoid overfitting

\[
\theta^* \leftarrow \arg \min_{\theta} \sum_{(x,y) \subseteq (X,Y)} \ell(y, a_L(x; \theta_1, \ldots, L)) + \lambda \Omega(\theta)
\]

• Possible regularization methods
  – $l_2$-regularization
  – $l_1$-regularization
  – Dropout
**$l_2$-regularization**

- Most important (or most popular) regularization
  \[
  \theta^* \leftarrow \arg\min_{\theta} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, \ldots, L)) + \frac{\lambda}{2} \sum_l \|\theta_l\|^2
  \]

- The $l_2$-regularization can pass inside the gradient descend update rule
  \[
  \theta^{(t+1)} = \theta^{(t)} - \eta_t (\nabla_\theta \mathcal{L} + \lambda \theta_l) \Rightarrow \\
  \theta^{(t+1)} = (1 - \lambda \eta_t) \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}
  \]

- $\lambda$ is usually about $10^{-1}, 10^{-2}$
\( l_1 \)-regularization

- \( l_1 \)-regularization is one of the most important techniques

\[
\theta^* \leftarrow \arg \min_{\theta} \sum_{(x,y) \subseteq (X,Y)} \ell(y, a_L(x; \theta_1, ..., L)) + \frac{\lambda}{2} \sum_l \|\theta_l\|
\]

- Also \( l_1 \)-regularization passes inside the gradient descend update rule

\[
\theta^{(t+1)} = \theta^{(t)} - \lambda \eta_t \left( \frac{\theta^{(t)}}{|\theta^{(t)}|} \right) - \eta_t \nabla \theta \mathcal{L}
\]

- \( l_1 \)-regularization → sparse weights
- \( \lambda \uparrow \rightarrow \) more weights become 0
Data augmentation [Krizhevsky2012]

Original

Flip

Contrast

Random crop

Tint
Noise as a regularizer

• Suppose we add Gaussian noise to the inputs.
  – The variance of the noise is amplified by the squared weight before going into the next layer.

• In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.

• This makes an additive contribution to the squared error.
  – So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.

Not exactly equivalent to using an L2 weight penalty.
Multi-task Learning

• Improving generalization by pooling the examples arising out of several tasks.
• Different supervised tasks share the same input $x$, as well as some intermediate-level representation $h(\text{shared})$
  – Task-specific parameters
  – Generic parameters (shared across all the tasks)
Early stopping

- Start with small weights and stop the learning before it overfits.
- Think early stopping as a very efficient hyperparameter selection.
  - The number of training steps is just another hyperparameter.

\[ \hat{y} = y_i \text{ where } i = \arg \min ||X_i, x||^2 \]

The algorithm can also be generalized to distance metrics other than the L2 norm, such as learned distance metrics (Goldberger et al., 2005). If the algorithm is allowed to break ties by averaging the \( y_i \) values for all \( X_i, x \) that are tied for nearest, then this algorithm is able to achieve the minimum possible training error (which might be greater than zero, if two identical inputs are associated with different outputs) on any regression dataset.
**Model Ensembles**

- Train several different models separately, then have all of the models vote on the output for test examples.
- Different models will usually not make all the same errors on the test set.
- Usually ~2% gain!
Model Ensembles

• We 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:

```
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
```
Dropout

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

[Srivastava et al., 2014]
Waaaait a second...
How could this possibly be a good idea?
Waaaait 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
- cat score

How could this possibly be a good idea?
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).
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).

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

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

=> We must scale the activations so that for each neuron:
output at test time = expected output at training time
Dropout Summary

""" 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):
    """ 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)
```

don't in forward pass

```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
```

scale at test time
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

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

test time is unchanged!
Optimization
Training a neural network, main loop:

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Training a neural network, main loop:

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```

simple gradient descent update now: complicate.
Gradients

- When we write $\nabla_W L(W)$, we mean the vector of partial derivatives wrt all coordinates of $W$:
  \[
  \nabla_W L(W) = \left[ \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \ldots, \frac{\partial L}{\partial W_m} \right]^T
  \]
  where $\frac{\partial L}{\partial W_i}$ measures how fast the loss changes vs. change in $W_i$

- **In figure:** loss surface is blue, gradient vectors are red:

- When $\nabla_W L(W) = 0$, it means all the partials are zero, i.e. the loss is not changing in any direction.

- Note: arrows point out from a minimum, in toward a maximum

Slide adapted from John Canny 84
Optimization

• Visualizing gradient descent in one dimension:

• The regions where gradient descent converges to a particular local minimum are called **basins of attraction**.
Local Minima

• Since the optimization problem is non-convex, it probably has local minima.

• This kept people from using neural nets for a long time, because they wanted guarantees they were getting the optimal solution.

• But are local minima really a problem?
  – Common view among practitioners: yes, there are local minima, but they’re probably still pretty good.
    • Maybe your network wastes some hidden units, but then you can just make it larger.
  – It’s very hard to demonstrate the existence of local minima in practice.
  – In any case, other optimization-related issues are much more important.
Saddle Points

• At a **saddle point**, \( \frac{\partial L}{\partial W} = 0 \) even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

• When would saddle points be a problem?
  – If we’re exactly on the saddle point, then we’re stuck.
  – If we’re slightly to the side, then we can get unstuck.
Saddle Points

• At a **saddle point**, \( \frac{\partial L}{\partial W} = 0 \) even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

• When would saddle points be a problem?
  – If we’re exactly on the saddle point, then we’re stuck.
  – If we’re slightly to the side, then we can get unstuck.

*Saddle points much more common in high dimensions!*

Y. Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS 2014
Plateaux

• A flat region is called a plateau. (Plural: plateaux)
Plateaux

• An important example of a plateau is a **saturated unit**. This is when it is in the flat region of its activation function.

• If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.

• If there is a ReLU unit whose input $z_i$ is always negative, the weight derivatives will be exactly 0. We call this a **dead unit**.
Loss surfaces in high-dimensional problems are very complicated!
Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration $k$

Require: Learning rate $\epsilon_k$
Require: Initial Parameter $\theta$

1: while stopping criteria not met do
2: Compute gradient estimate over $N$ examples:
3: $\hat{g} \leftarrow +\frac{1}{N} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
4: Apply Update: $\theta \leftarrow \theta - \epsilon \hat{g}$
5: end while

• Positive: Gradient estimates are stable
• Negative: Need to compute gradients over the entire training for one update
Gradient Descent
Gradient Descent
Gradient Descent
Gradient Descent
Gradient Descent
Gradient Descent
Stochastic Batch Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration $k$

Require: Learning rate $\epsilon_k$

Require: Initial Parameter $\theta$

1: while stopping criteria not met do
2: Sample example $(x^{(i)}, y^{(i)})$ from training set
3: Compute gradient estimate:
4: $\hat{g} \leftarrow +\nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$
5: Apply Update: $\theta \leftarrow \theta - \epsilon \hat{g}$
6: end while
Minibatching

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- Advantage: Computation time per update does not depend on number of training examples $N$

- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Stochastic Gradient Descent
Image credits: Alec Radford
Q: What is the trajectory along which we converge towards the minimum with SGD?
Q: What is the trajectory along which we converge towards the minimum with SGD?
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

SGD

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

while True:
    dx = compute_gradient(x)
    x += learning_rate * dx

SGD+Momentum

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
Momentum update

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]

\[ x_{t+1} = x_t - \alpha v_{t+1} \]
SGD vs Momentum

notice momentum overshooting the target, but overall getting to the minimum much faster.
SGD + Momentum

Momentum update

- momentum step
- actual step
- gradient step
Nesterov Momentum

Momentum update

Nesterov momentum update

"lookahead" gradient step (bit different than original)

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

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]

\[ x_{t+1} = x_t + v_{t+1} \]
Nesterov Momentum

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

Change of variables \( \tilde{x}_t = x_t + \rho v_t \) and rearrange:

\[ v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t) \]
\[ \tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho) v_{t+1} \]
\[ = \tilde{x}_t + v_{t+1} + \rho (v_{t+1} - v_t) \]

Annoying, usually we want update in terms of \( x_t, \nabla f(x_t) \):

```python
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```
nag = Nesterov Accelerated Gradient
AdaGrad update

```python
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

[Duchi et al., 2011]
AdaGrad update

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Q: What happens with AdaGrad?
Weights that receive high gradients will have their effective learning rate reduced, while weights that receive small updates will have their effective learning rate increased!
Q2: What happens to the step size over long time?

The adaptive learning scheme is monotonic, which is usually too aggressive and stops the learning process too early.
RMSProp

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

[Tieleman and Hinton, 2012]
Adaptive Moment Estimation (Adam) (incomplete, but close)

Looks a bit like RMSProp with momentum

[first code block]

momentum
AdaGrad / RMSProp

[Kingma and Ba, 2014]
Adam (full form)

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

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = compute gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

momentum
Bias correction
AdaGrad / RMSProp

[Kingma and Ba, 2014]
Adam (full form)

```python
def adam_optimizer(x, learning_rate, beta1, beta2, num_iterations):
    first_moment = 0
    second_moment = 0
    for t in range(num_iterations):
        dx = compute_gradient(x)
        first_moment = beta1 * first_moment + (1 - beta1) * dx
        second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
        first_unbias = first_moment / (1 - beta1 ** t)
        second_unbias = second_moment / (1 - beta2 ** t)
        x = x - learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

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

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

[Kingma and Ba, 2014]
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:**

\[
\alpha = \frac{\alpha_0}{(1 + kt)}
\]
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

![Graph showing learning rate decay over time](image)
First-Order Optimization

1) Use gradient form linear approximation
2) Step to minimize the approximation
Second-Order Optimization

1) Use gradient and Hessian (H) to form **quadratic** approximation
2) Step to the **minima** of the approximation
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:

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

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

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:

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

**Q2:** why is this impractical for training Deep Neural Nets?

notice:
no hyperparameters! (e.g. learning rate)
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.
Babysitting the Learning Process
Say we start with one hidden layer of 50 neurons:

- **Input layer**
  - CIFAR-10 images, 3072 numbers

- **Hidden layer**
  - 50 hidden neurons

- **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):
    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, 0.0)  # disable regularization
print(loss)
```

Loss ~2.3. "correct" for 10 classes. 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, 1e3)  # crank up regularization
print loss
```

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

Start with small regularization and find learning rate that makes the loss go down.
Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

Loss barely changing
Let's try to train now...

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)
Okay now lets try learning rate 1e6. What could possibly go wrong?

loss not going down: learning rate too low
Lets try to train now…

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…

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 Selection
Everything is a hyperparameter

- Network size/depth
- Small model variations
- Minibatch creation strategy
- Optimizer/learning rate

- Models are complicated and opaque, debugging can be difficult!
Cross-validation strategy

First 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

```
max_count = 100
for count in range(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)
```

```
Now run finer search...

```python
max_count = 100
for count in range(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: 8.688082e-04, reg: 1.349727e-02, (2 / 100)
    val_acc: 0.461000, lr: 1.028377e-04, reg: 1.226193e-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: 0.469000, 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: 3.905540e-02, (12 / 100)
    val_acc: 0.516000, lr: 8.639527e-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?
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.

Learning rate close to the edge, need more wider search!
Random Search vs. Grid Search

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

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

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

- **no gap**
  - => increase model capacity?
The Double Descent Phenomenon

- Test error follows the traditional U-shaped curve in the under-parameterized case and monotonically decreases in the over-parameterized case.

(Neal et al., 2018), (Spigler et al., 2018), (Geiger et al., 2019), (Belkin et al., 2019)
Visualization

- Check gradients numerically by finite differences
- Visualize features (features need to be uncorrelated) and have high variance

• Good training: hidden units are sparse across samples

From Marc'Aurelio Ranzato, CVPR 2014 tutorial
Visualization

• Check gradients numerically by finite differences

• Visualize features (features need to be uncorrelated) and have high variance

• Bad training: many hidden units ignore the input and/or exhibit strong correlations

From Marc'Aurelio Ranzato, CVPR 2014 tutorial
Visualization

• Check gradients numerically by finite differences

• Visualize features (features need to be uncorrelated) and have high variance

• Visualize parameters: learned features should exhibit structure and should be uncorrelated and are uncorrelated

From Marc’Aurelio Ranzato, CVPR 2014 tutorial
Take Home Messages
Optimization Tricks

• SGD with momentum, batch-normalization, and dropout usually works very well

• Pick learning rate by running on a subset of the data
  • Start with large learning rate & divide by 2 until loss does not diverge
  • Decay learning rate by a factor of ~100 or more by the end of training

• Use ReLU nonlinearity

• Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.

From Marc'Aurelio Ranzato, CVPR 2014 tutorial
Ways To Improve Generalization

• Weight sharing (greatly reduce the number of parameters)
• Dropout
• Weight decay (L2, L1)
• Sparsity in the hidden units
Babysitting

• Check gradients numerically by finite differences

• Visualize features (features need to be uncorrelated) and have high variance

• Visualize parameters: learned features should exhibit structure and should be uncorrelated and are uncorrelated

• Measure error on both training and validation set

• Test on a small subset of the data and check the error $\rightarrow 0$. 

From Marc'Aurelio Ranzato, CVPR 2014 tutorial
Next lecture: Convolutional Neural Networks