Lecture 8:
Training Neural Networks, Part 2
Administrative:

- A1 grades released: Check Piazza for regrade policy
- Project proposal due yesterday
- A2 due Wednesday 5/1
Last time: Activation Functions

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

\[
\text{tanh}
\]
\[ \tanh(x) \]

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

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

\[
\text{Maxout}
\]
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

\[
\text{ELU}
\]
\[ \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
\end{cases} \]
Last time: Activation Functions

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

\textbf{tanh}
\[ \tanh(x) \]

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

Good default choice

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

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

ELU
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Last time: Weight Initialization

Initialization too small:
Activations go to zero, gradients also zero,
No learning =(  

Initialization too big:
Activations saturate (for tanh),
Gradients zero, no learning =(  

Initialization just right:
Nice distribution of activations at all layers,
Learning proceeds nicely
Last time: Data Preprocessing

- **original data**
- **zero-centered data**
- **normalized data**
Last Time: Batch Normalization

**Input:** \( x : N \times D \)

Per-channel mean, shape is D

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

Per-channel var, shape is D

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

Normalized x, Shape is N x D

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}
\]

Output, Shape is N x D

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]

[ioffe and Szegedy, 2015]
Today

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters
Optimization

```python
# Vanilla Gradient Descent

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

![Gradient descent diagram](image)
Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do?

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.
Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do? Very slow progress along shallow dimension, jitter along steep direction.

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
Optimization: Problems with SGD

What if the loss function has a local minima or saddle point?

Saddle points much more common in high dimension

Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014
Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) \]
**SGD + Momentum**

**SGD**

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

**SGD+Momentum**

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

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum

SGD+Momentum

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

\text{while True:}
\begin{align*}
\text{dx} &= \text{compute\_gradient}(x) \\
\text{vx} &= \rho \star \text{vx} - \text{learning\_rate} \star \text{dx} \\
x &= + \text{vx}
\end{align*}

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum

Local Minima  Saddle points

Poor Conditioning

Gradient Noise
SGD+Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Nesterov Momentum

**Momentum update:**

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

**Nesterov Momentum**

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction
Nesterov Momentum

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

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

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

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

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

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

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
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 \( x_t, \nabla f(x_t) \)

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction.
Nesterov Momentum

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

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

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

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

```python
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```
Nesterov Momentum

- SGD
- SGD+Momentum
- Nesterov
AdaGrad

```
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

“Per-parameter learning rates” or “adaptive learning rates”

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011
Q: What happens with AdaGrad?

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

```
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? Progress along “steep” directions is damped; progress along “flat” directions is accelerated.
AdaGrad

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

Q2: What happens to the step size over long time?
AdaGrad

\[
\text{grad\_squared} = 0 \\
\text{while True:} \\
\quad \text{dx} = \text{compute\_gradient}(x) \\
\quad \text{grad\_squared} += \text{dx} \times \text{dx} \\
\quad x -= \text{learning\_rate} \times \text{dx} / (\text{np.sqrt(grad\_squared)} + 1e-7)
\]

Q2: What happens to the step size over long time? Decays to zero
RMSProp: “Leaky AdaGrad”

Adagrad

\[
\text{grad\_squared} = 0 \\
\text{while True:} \\
\quad dx = \text{compute\_gradient}(x) \\
\quad \text{grad\_squared} += dx \cdot dx \\
\quad x -= \text{learning\_rate} \cdot dx / (\text{np.sqrt}(	ext{grad\_squared}) + 1e-7)
\]

RMSProp

\[
\text{grad\_squared} = 0 \\
\text{while True:} \\
\quad dx = \text{compute\_gradient}(x) \\
\quad \text{grad\_squared} = \text{decay\_rate} \cdot \text{grad\_squared} + (1 - \text{decay\_rate}) \cdot dx \cdot dx \\
\quad x -= \text{learning\_rate} \cdot dx / (\text{np.sqrt}(	ext{grad\_squared}) + 1e-7)
\]

Tieleman and Hinton, 2012
RMSProp

- SGD
- SGD+Momentum
- RMSProp
Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Adam (almost)

```python
first moment = 0
second moment = 0
while True:
    dx = compute_gradient(x)
    first moment = beta1 * first moment + (1 - beta1) * dx
    second moment = beta2 * second moment + (1 - beta2) * dx * dx
    x -= learning rate * first moment / (np.sqrt(second moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(1, 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

Bias correction for the fact that first and second moment estimates start at zero

Adam (full form)

```
def adam(x, num_iterations):
    first_moment = 0
    second_moment = 0
    for t in range(1, 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

Bias correction for the fact that first and second moment estimates start at zero

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

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.

Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time.
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

\[
\alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right)
\]

\(\alpha_0\) : Initial learning rate  
\(\alpha_t\) : Learning rate at epoch \(t\)  
\(T\) : Total number of epochs

---

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \( \alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T)) \)

---

- Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
- Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
- Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018

\( \alpha_0 \) : Initial learning rate  
\( \alpha_t \) : Learning rate at epoch t  
\( T \) : Total number of epochs
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T)) \]

**Linear:** \[ \alpha_t = \alpha_0 (1 - t/T) \]

\( \alpha_0 \): Initial learning rate  
\( \alpha_t \): Learning rate at epoch \( t \)  
\( T \): Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

- **Cosine:** \( \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right) \)
- **Linear:** \( \alpha_t = \alpha_0 (1 - t/T) \)
- **Inverse sqrt:** \( \alpha_t = \frac{\alpha_0}{\sqrt{t}} \)

\( \alpha_0 \): Initial learning rate  
\( \alpha_t \): Learning rate at epoch \( t \)  
\( T \): Total number of epochs

---

Vaswani et al, “Attention is all you need”, NIPS 2017
High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this.

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N.

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017
First-Order Optimization

Loss

w1
First-Order Optimization

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

(1) Use gradient and Hessian to form quadratic approximation
(2) Step to the minima of the approximation
Second-Order Optimization

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:

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

Q: Why is this bad for deep learning?
Second-Order Optimization

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:

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

Hessian has \(O(N^2)\) elements
Inverting takes \(O(N^3)\)
\(N = \) (Tens or Hundreds of) Millions

**Q: Why is this bad for deep learning?**
Second-Order Optimization

- 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.

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

- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017
In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)
Beyond Training Error

Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?
Early Stopping: Always do this

Stop training the model when accuracy on the validation set decreases.
Or train for a long time, but always keep track of the model snapshot
that worked best on val.
Model Ensembles

1. Train multiple independent models
2. At test time average their results
   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al. “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al. “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```python
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
```

How to improve single-model performance?

Regularization
Regularization: Add term to loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W) \]

In common use:

**L2 regularization**
\[ R(W) = \sum_k \sum_l W_{k,l}^2 \quad \text{(Weight decay)} \]

**L1 regularization**
\[ R(W) = \sum_k \sum_l |W_{k,l}| \]

**Elastic net (L1 + L2)**
\[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
Regularization: Dropout

In each forward pass, randomly set some neurons to zero. The probability of dropping is a hyperparameter; 0.5 is common.

Regularization: Dropout

\[ p = 0.5 \]  # 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)
```

Example forward pass with a 3-layer network using dropout
Regularization: Dropout
How can this possibly be a good idea?

Forces the network to have a redundant representation;
Prevents co-adaptation of features

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

X
X
X

X

cat score
Regularization: Dropout

How can 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

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...
Dropout: Test time

Dropout makes our output random!

Want to “average out” the randomness at test-time

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

But this integral seems hard …
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.

At test time we have:

\[ \mathbb{E}[a] = w_1 x + w_2 y \]
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)\,dz \]

Consider a single neuron.

At test time we have:

\[ E[a] = w_1 x + w_2 y \]

During training we have:

\[
E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)
\]
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)\,dz \]

Consider a single neuron.

At test time we have:
\[ E[a] = w_1 x + w_2 y \]

During training we have:
\[
E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\
+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\
= \frac{1}{2}(w_1 x + w_2 y)
\]

At test time, **multiply** by dropout probability
Dropout: Test time

```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)

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
```

drop in forward pass

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
```

test time is unchanged!
Regularization: A common pattern

**Training**: Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing**: Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz \]
Regularization: A common pattern

**Training**: Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing**: Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz \]

**Example**: Batch Normalization

**Training**: Normalize using stats from random minibatches

**Testing**: Use fixed stats to normalize
Regularization: Data Augmentation

Load image and label

“cat”

Compute loss

CNN

This image by Nikita is licensed under CC-BY 2.0
Regularization: Data Augmentation

Load image and label

“cat”

Transform image

Compute loss

CNN
Data Augmentation

Horizontal Flips
Data Augmentation
Random crops and scales

**Training**: sample random crops / scales

ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch
Data Augmentation
Random crops and scales

**Training**: sample random crops / scales
ResNet:
1. Pick random \(L\) in range \([256, 480]\)
2. Resize training image, short side = \(L\)
3. Sample random \(224 \times 224\) patch

**Testing**: average a fixed set of crops
ResNet:
1. Resize image at 5 scales: \(\{224, 256, 384, 480, 640\}\)
2. For each size, use 10 \(224 \times 224\) crops: 4 corners + center, + flips
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness

More Complex:
1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in [Krizhevsky et al. 2012], ResNet, etc)
Data Augmentation
Get creative for your problem!

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)
Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
Regularization: DropConnect

**Training**: Drop connections between neurons (set weights to 0)

**Testing**: Use all the connections

**Examples**:  
Dropout  
Batch Normalization  
Data Augmentation  
DropConnect

---

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Regularization: Fractional Pooling

**Training**: Use randomized pooling regions

**Testing**: Average predictions from several regions

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling

Graham, "Fractional Max Pooling", arXiv 2014
Regularization: Stochastic Depth

**Training**: Skip some layers in the network

**Testing**: Use all the layers

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth

Regularization: Cutout

Training: Set random image regions to zero
Testing: Use full image

Examples:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017

Works very well for small datasets like CIFAR, less common for large datasets like ImageNet
Regularization: Mixup

**Training**: Train on random blends of images

**Testing**: Use original images

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout
- Mixup

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Target label: cat: 0.4, dog: 0.6

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018
Regularization

**Training:** Add random noise  
**Testing:** Marginalize over the noise

**Examples:**

- Dropout  
- Batch Normalization  
- Data Augmentation  
- DropConnect  
- Fractional Max Pooling  
- Stochastic Depth  
- Cutout  
- Mixup  

- Consider dropout for large fully-connected layers  
- Batch normalization and data augmentation almost always a good idea  
- Try cutout and mixup especially for small classification datasets
Choosing Hyperparameters
(without tons of GPUs)
Choosing Hyperparameters

**Step 1:** Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. log(C) for softmax with C classes
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization
Loss explodes to Inf or NaN? LR too high, bad initialization
Choosing Hyperparameters

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4
Choosing Hyperparameters

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down
**Step 4:** Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0
Choosing Hyperparameters

**Step 1:** Check initial loss
**Step 2:** Overfit a small sample
**Step 3:** Find LR that makes loss go down
**Step 4:** Coarse grid, train for ~1-5 epochs
**Step 5:** Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Losses may be noisy, use a scatter plot and also plot moving average to see trends better.
Bad initialization a prime suspect
Loss plateaus: Try learning rate decay
Loss was still going down when learning rate dropped, you decayed too early!
Accuracy still going up, you need to train longer.

Accuracy

Train

Val

time
Huge train / val gap means overfitting! Increase regularization, get more data.
No gap between train / val means underfitting: train longer, use a bigger model
Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Step 7: GOTO step 5
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

This image by Paolo Guereta is licensed under CC-BY 2.0
Cross-validation
“command center”
Random Search vs. Grid Search

Grid Layout

Random Layout

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017
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 updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so
Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules

- Improve your test error:
  - Regularization
  - Choosing Hyperparameters
Next time: CNN Architecture Design