Lecture 9: Regularization and optimization

Announcements:

- HW #3 is due ~~tonight~~ Friday. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

- Midterm is in 2 weeks on Feb 21, 2024. Past exams are uploaded to Bruin Learn (under "Modules" —> "past exams"). This year, we will allow 4 cheat sheets (8.5 x 11" paper) that can be filled front and back (8 sides total). The exam is otherwise closed book and closed notes.

- MT review session: Thursday, Feb 15, 2024, 6-9pm @ Young CS50.

  There will be a Zoom link 4 recording.

  TAs will not take questions over Zoom.

What is regularization?

Regularizations

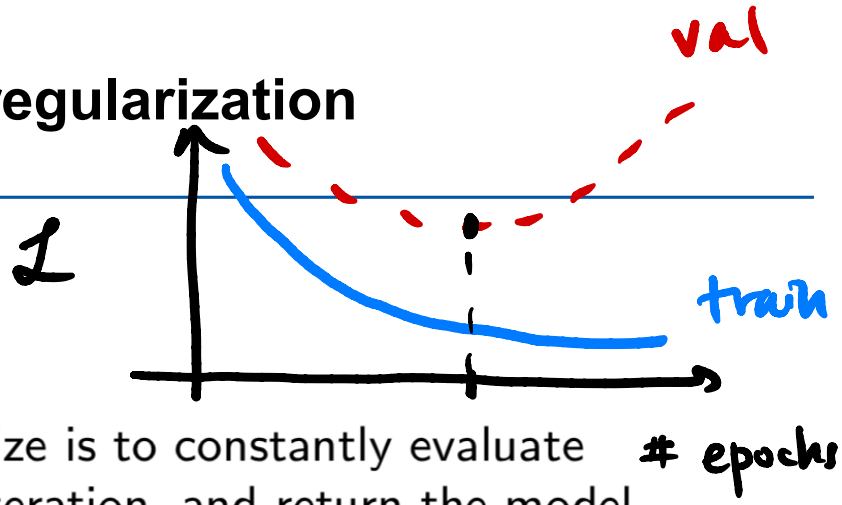
Regularizations are used to improve model generalization. Goodfellow, Bengio, and Courville define regularization in the following way (Deep Learning, p. 221):

> *[Regularization is] any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

In this manner, regularization is used to improve the *generalizability* of the model. Other intuitions:

- Regularization tends to increase the estimator bias while reducing the estimator variance.
- Regularization can be seen as a way to prevent overfitting.
- A common problem is in picking the model size and complexity. It may be appropriate to simply choose a large model that is regularized appropriately.

A simple example of regularization



A simple example: stopping early

One straightforward (and popular) way to regularize is to constantly evaluate the training and validation loss on each training iteration, and return the model with the lowest validation error.

- Requires caching the lowest validation error model.
- Training will stop when after a pre-specified number of iterations, no model has decreased the validation error.
- The number of training steps can be thought of as another hyperparameter.
- Validation set error can be evaluated in parallel to training.
- It doesn't require changing the model or cost function.
- The following is beyond the scope of the class – but in case curious, early stopping can be seen as a form of L^2 regularization (to be discussed in the next slides). See Goodfellow et al., *Deep Learning*, p. 242-5 for an indepth discussion.

A regularization familiar to you all: parameter norm penalties

You guys have already implemented some regularization on the HWs through parameter norm penalties.

These are not specific to neural networks. These are commonly used, e.g., even in linear regression, where specific penalty norms have their own names (e.g., Tikhonov regularization / ridge regression).

$$\propto \|W\|_F$$

Regularization via parameter norm penalties

A common (and simple to implement) type of regularization is to modify the cost function with a *parameter norm penalty*. This penalty is typically denoted as $\Omega(\theta)$ and results in a new cost function of the form:

1

$$J(\theta; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta)$$

with $\alpha \geq 0$. A few things to note:

- α is a hyperparameter that weights the contribution of the norm penalty. When $\alpha = 0$, there is no regularization. When $\alpha \to \infty$, the cost function is irrelevant and the model will set the parameters to minimize $\Omega(\theta)$.
- The choice of α can strongly affect generalization performance.

L2 regularization

L^2 regularization

A common form of parameter norm regularization is to penalize the size of the weights (L^2 regularization). This is also commonly called "ridge regression" or "Tikhonov regularization." This promotes models with parameters that are closer to 0 (and hence, colloquially speaking, "simpler"). If $\mathbf{w}$ are the model parameters to be regularized, then L^2 regularization sets:

$$\Omega(\theta) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}\|w\|^2$$

$$\Omega(\theta) = \frac{1}{2}\|W\|_F^2$$

Intuitively, to prevent $\Omega(\theta)$ from getting large, L^2 regularization will cause the weights $\mathbf{w}$ to have small norm.

$$\frac{\partial \Omega(\theta)}{\partial W} = W$$

0.1 0.01

$x^{(1)}$ $x^{(2)}$

y

$1 = 100 \cdot x^{(1)} - 900 \cdot x^{(2)}$

L2 regularization

L^2 regularization (cont)

More formally, when using L^2 regularization, the new cost function is:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

with corresponding gradient:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

The gradient step is:

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

This formalizes the intuition that L^2 regularization will shrink the weights, $\mathbf{w}$, before performing the usual gradient update.

No param. norm penalty

$w \leftarrow w - \varepsilon \nabla_w J$

With param. norm

$w \leftarrow w - \varepsilon \nabla_w \tilde{J}$

$w \leftarrow w - \varepsilon \left(\nabla_w J + \alpha w \right)$

$w \leftarrow (1 - \varepsilon\alpha) w - \varepsilon \nabla_w J$

Other places you may have seen L2 regularization (NOT tested)

Other equivalent statements of L^2 regularization

While we won't discuss these at length in class, it may be worthwhile to work out these equivalences:

- L^2 regularization is equivalent to maximum a-posteriori inference, where the prior on the parameters has a unit Gaussian distribution, i.e.,

$$\mathbf{w} \sim \mathcal{N}(0, \frac{1}{\alpha}\mathbf{I})$$

- When performing L^2 regularization, the component of $\mathbf{w}$ aligned with the ith eigenvector of the Hessian is rescaled by a factor

$$\frac{\lambda_i}{\lambda_i + \alpha}$$

- In linear regression, the least squares solution $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ becomes:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

scaling the variance of each input feature. The dimensions of $\mathbf{X}^T\mathbf{X}$ that are large (i.e., high variance) aren't affected as much, while those dimensions where $\mathbf{X}^T\mathbf{X}$ is small are.

L2 regularization

$\frac{\alpha}{2} \|w\|^2$

Extensions of L^2 regularization

Other related forms of regularization include:

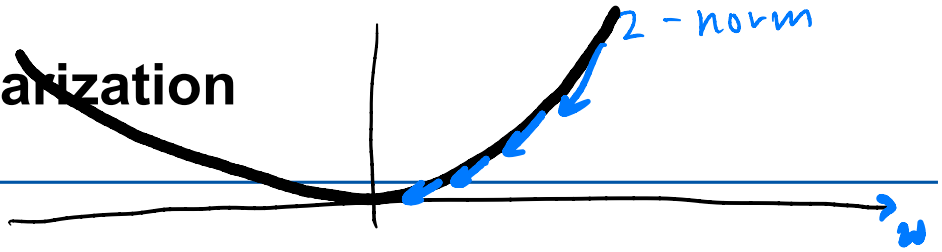
- Instead of a soft constraint that $\mathbf{w}$ be small, one may have prior knowledge that $\mathbf{w}$ is close to some value, $\mathbf{b}$. Then, the regularizer may take the form:

$$\Omega(\theta) = \|\mathbf{w} - \mathbf{b}\|_2^2$$

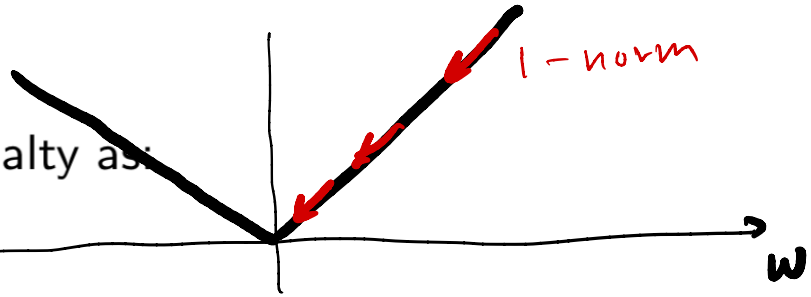
- One may have prior knowledge that two parameters, $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$, ought be close to each other. Then, the regularizer may take the form:

$$\Omega(\theta) = \left\|\mathbf{w}^{(1)} - \mathbf{w}^{(2)}\right\|_2^2$$

L1 regularization



L^1 regularization



L^1 regularization defines the parameter norm penalty as:

$$\begin{aligned}\Omega(\theta) &= \|\mathbf{w}\|_1 \\ &= \sum_i |w_i|\end{aligned}$$

Intuitively, this penalty also causes the weights to be small. However, because the subgradient of $\|\mathbf{w}\|_1$ is $\text{sign}(\mathbf{w})$, the gradient is the same regardless of the size of $\mathbf{w}$. (Contrast this to L^2 regularization, where the size of $\mathbf{w}$ matters.) Empirically, this typically results in sparse solutions where $w_i = 0$ for several i.

- This may be used for *feature selection*, where features corresponding to zero weights may be discarded.
- L^1 regularization is equivalent to maximum a-posteriori inference where the prior on the parameters has an isotropic Laplace distribution, i.e.,

$$w_i \sim \text{Laplace}\left(0, \frac{1}{\alpha}\right)$$

L1 regularization on the units

Sparse representations

Instead of having sparse parameters (i.e., elements of $\mathbf{w}$ being sparse), it may be appropriate to have sparse *representations*. Imagine a hidden layer of activity, $\mathbf{h}^{(i)}$. To achieve a sparse representation, one may set:

$$\Omega(\mathbf{h}^{(i)}) = \left\| \mathbf{h}^{(i)} \right\|_1$$

Dataset augmentation

Original image:

Dataset augmentation

Original image:

Flipped image:

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Adjust brightness

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Adjust brightness

Lens correction

Dataset augmentation

Original image:

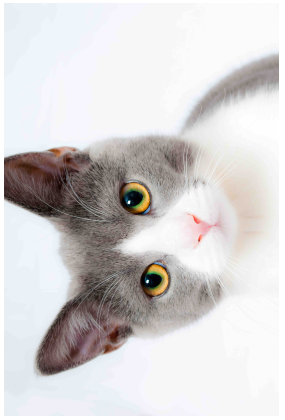
Flipped image:

Cropped image:

Adjust brightness

Lens correction

Rotate



Dataset augmentation

GoogLeNet 2014

2012 : ~16%

There are various heuristics to keeping the input size the same.

2014 : ~ 7%

You can be creative for dataset augmentation.

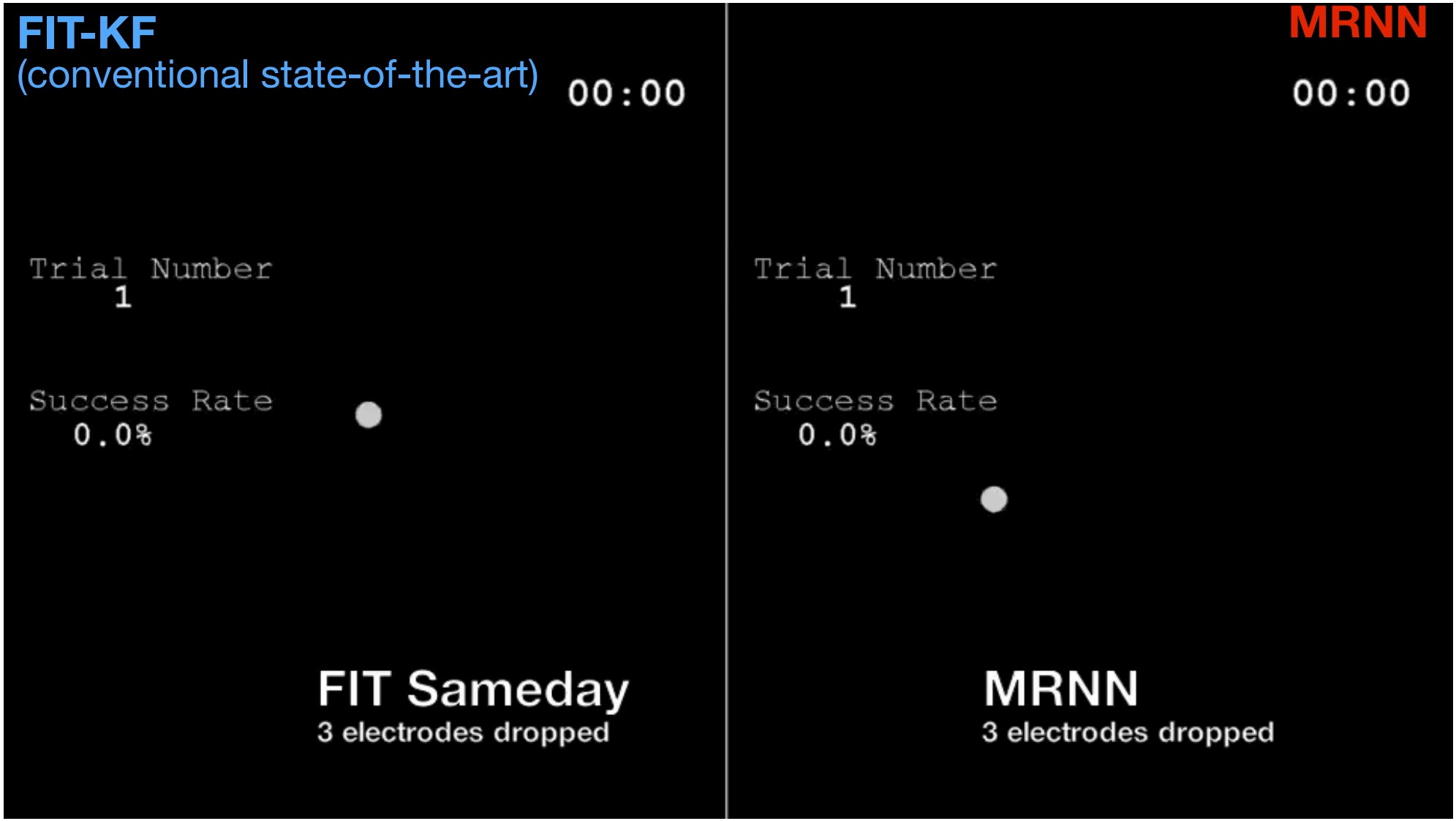
2. During testing, we adopted a more aggressive cropping approach than that of Krizhevsky et al. [9]. Specifically, we resized the image to 4 scales where the shorter dimension (height or width) is 256, 288, 320 and 352 respectively, take the left, center and right square of these resized images (in the case of portrait images, we take the top, center and bottom squares). For each square, we then take the 4 corners and the center 224×224 crop as well as the square resized to 224×224, and their mirrored versions. This leads to 4×3×6×2 = 144 crops per image. A similar ap-

Number of models	Number of Crops	Cost	Top-5 error	compared to base
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Szegedy et al., CVPR 2015

Dataset augmentation

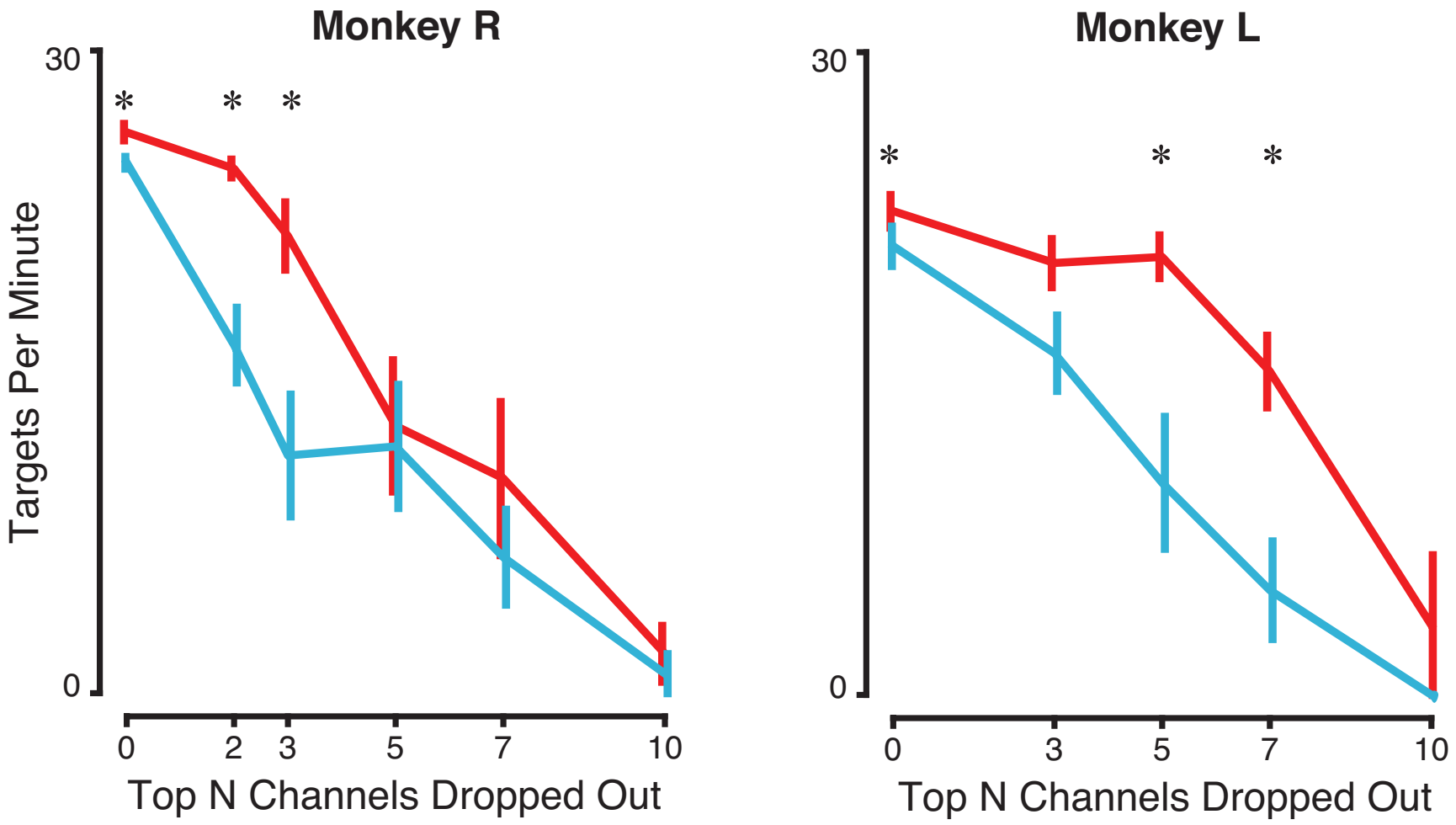
Example from brain-machine interfaces:



(back-to-back comparisons during the same experiment)

Dataset augmentation

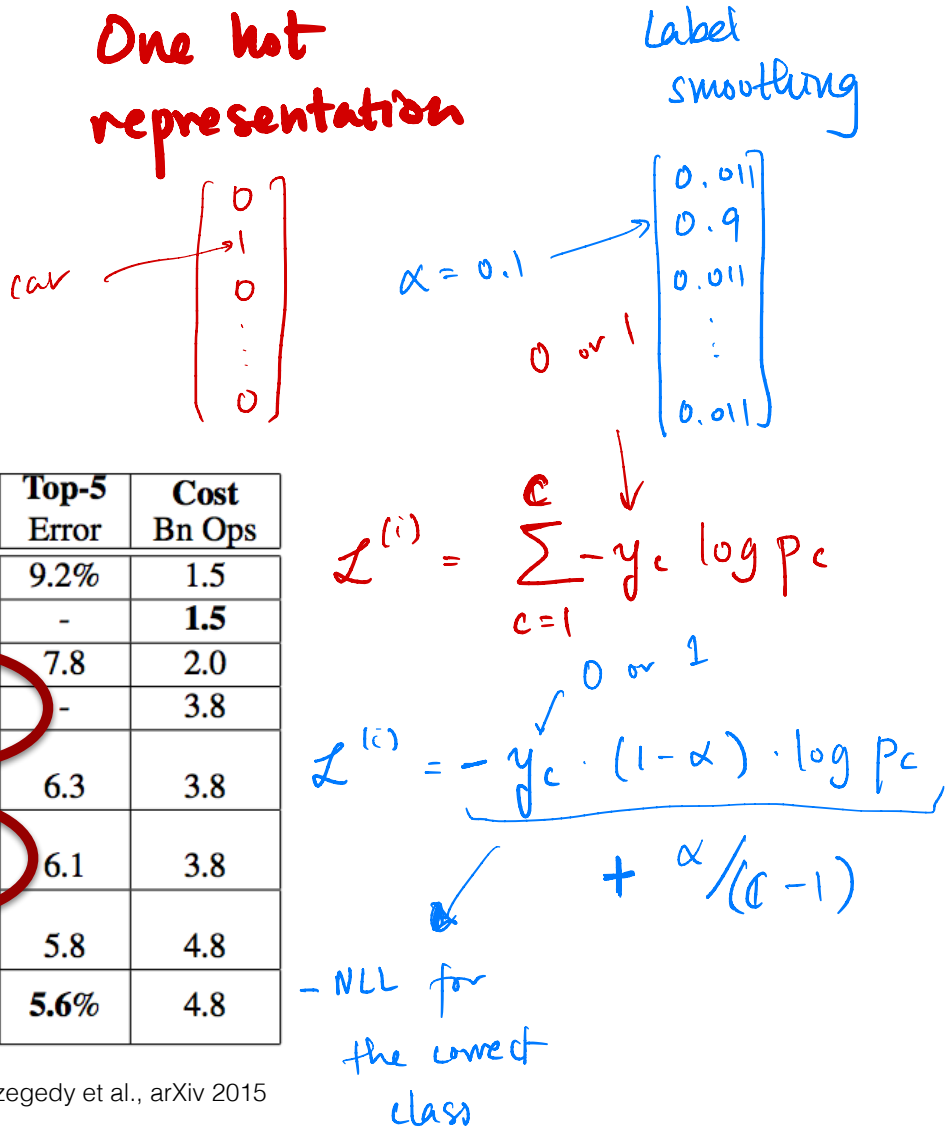
Example from brain-machine interfaces:



Dataset augmentation

Other types of dataset augmentation:

- Inject noise into the network
- Label smoothing



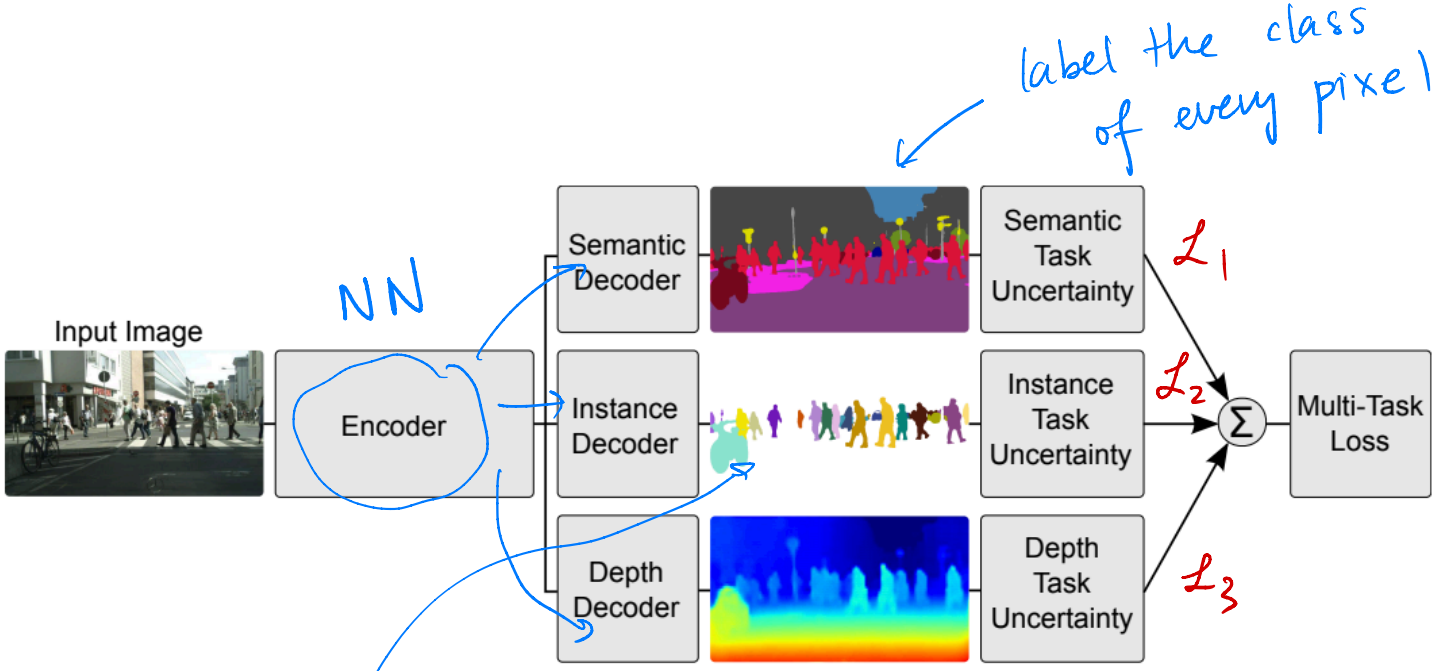
Network	Top-1 Error	Top-5 Error	Cost Bn Ops
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	**1.5**
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7 × 7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	**21.2%**	**5.6%**	4.8

Szegedy et al., arXiv 2015

Multitask learning

Another way to improve generalization is by having the model be trianed to perform multiple tasks. This represents the prior belief that multiple tasks share common factors to explain variations in the data.

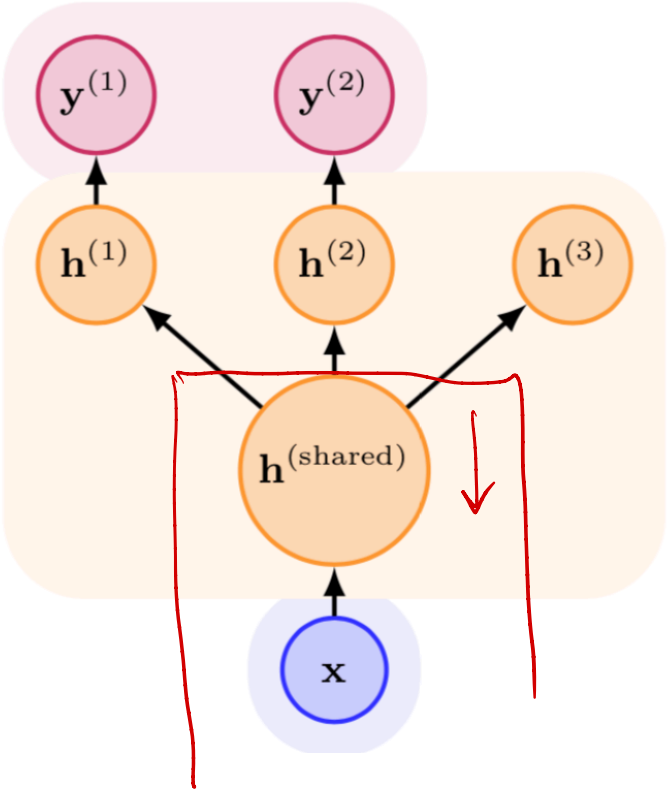
Multitask learning



Kendall et al., arXiv 2017

Multitask learning

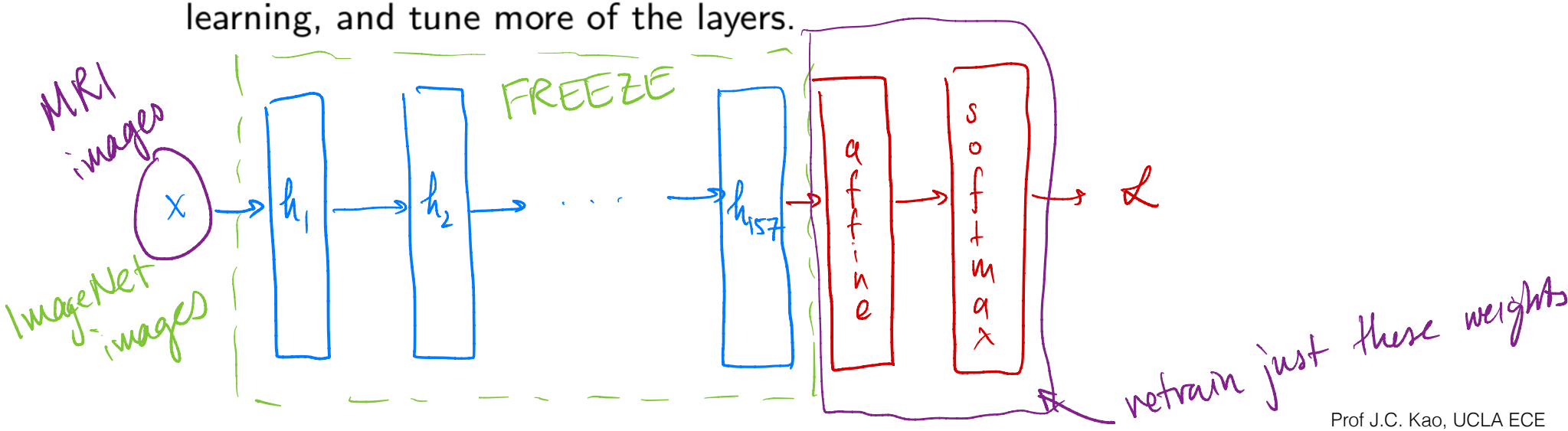
- The entire model need not be shared across different tasks.
- Here, $\mathbf{h}^{\text{shared}}$ captures common features that are then used by task-specific layers to predict $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$.
- $\mathbf{h}^{(3)}$ could represent a feature for unsupervised learning.



Transfer learning

We'll discuss this more in the convolutional neural networks lecture, but a related idea is to take neural networks trained in one context and use them in another with little additional training.

- The idea is that if the tasks are similar enough, then the features at later layers of the network ought to be good features for this new task.
- If little training data is available to you, but the tasks are similar, all you may need to do is train a new linear layer at the output of the pre-trained network.
- If more data is available, it may still be a good idea to use transfer learning, and tune more of the layers.



Ensemble methods

One way to get a boost in performance for very little cognitive work is to use ensemble methods.

The approach is:

1. Train multiple different models
2. Average their results together at test time.

This almost always increases performance by substantial amounts (e.g., a few percentage improvement in testing).

Ensemble methods

- The basic intuition between ensemble methods is that if models are independent, they will usually not all make the same errors on the test set.
- With k independent models, the average model error will decrease by a factor $\frac{1}{k}$. Denoting ϵ_i to be the error of model i on an example, and assuming $\mathbb{E}\epsilon_i = 0$ as well as that the statistics of this error is the same across all models,

$\mathbb{E}[\varepsilon_i \varepsilon_j] = \mathbb{E}[\varepsilon_i]\mathbb{E}[\varepsilon_j]$ if $\varepsilon_i \perp\!\!\!\perp \varepsilon_j$

$\mathbb{E}[\varepsilon_i^2]$

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] = \frac{1}{k^2}\sum_{i=1}^{k}\mathbb{E}\epsilon_i^2$$

$$= \frac{1}{k}\mathbb{E}\epsilon_i^2$$

- If the models are not independent, it can be shown that:

$$\frac{1}{k}\mathbb{E}\epsilon_i^2 + \frac{k-1}{k}\mathbb{E}[\epsilon_i\epsilon_j]$$

worst case

$\varepsilon_i = \varepsilon_j$

which is equal to $\mathbb{E}\epsilon_i^2$ only when the models are perfectly correlated.

Ensemble methods

One implementation of ensemble methods is via bagging.

Bagging

Bagging stands for bootstrap aggregating. It is an ensemble method for regularization. The procedure is as follows:

- Construct k datasets using the bootstrap (i.e., set a data size, N, and draw with replacement from the original dataset to get N samples; do this k times).
- Train k different models using these k datasets.

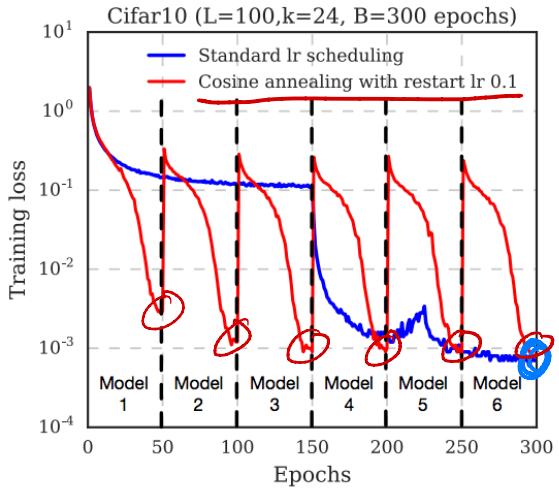
A few notes:

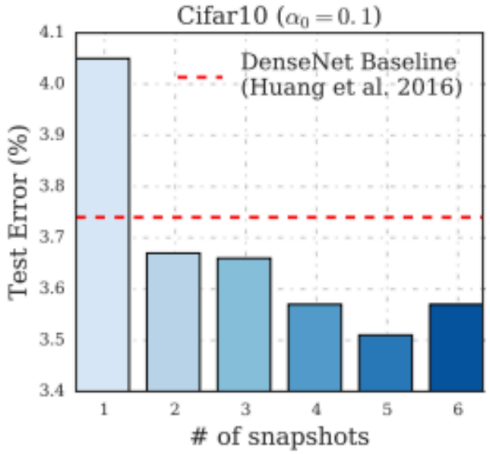
- In practice, neural networks reach a wide variety of solutions given different initializations, hyperparameters, etc., and so in practice even if they are trained from the same dataset, they tend to produce partially independent errors.
- While model averaging is powerful, it is expensive for neural networks, since the time to train models can be very large.

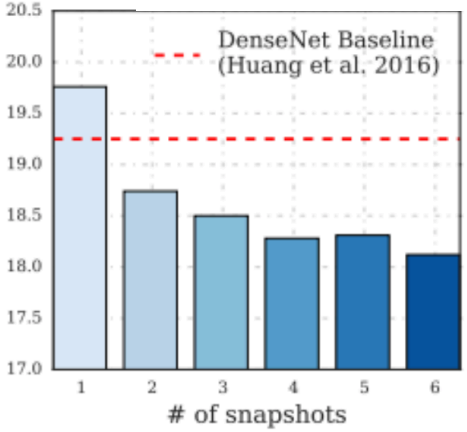
Ensemble methods

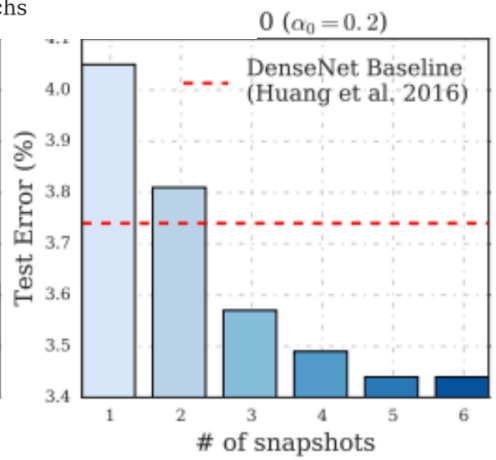
Ways to get around computational expense?

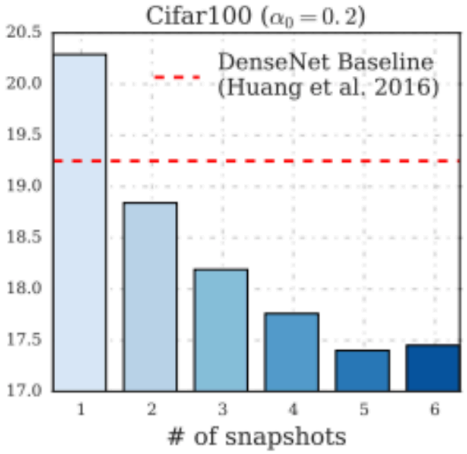
You could take snapshots of the model at different local minima and average them together. See Huang, Li et al., ICLR 2017.











Dropout

p : probability of keeping a neuron in a layer

Goodfellow

Dropout is a computationally inexpensive yet effective method for generalization. It can be viewed as approximating the bagging procedure for exponentially many models, while only optimizing a single set of parameters. The following steps describe the dropout regularizer:

- On a given training iteration, sample a binary mask (i.e., each element is 0 or 1) for all input and hidden units in the network.
  - The Bernoulli parameter, p, is a hyperparameter.
  - Typical values are that $p = 0.8$ for input units and $p = 0.5$ for hidden units.
- Apply (i.e., multiply) the mask to all units. Then perform the forward pass and backwards pass, and parameter update.
- In *prediction*, multiply each hidden unit by the parameter of its Bernoulli mask, p.

h

100 neurons

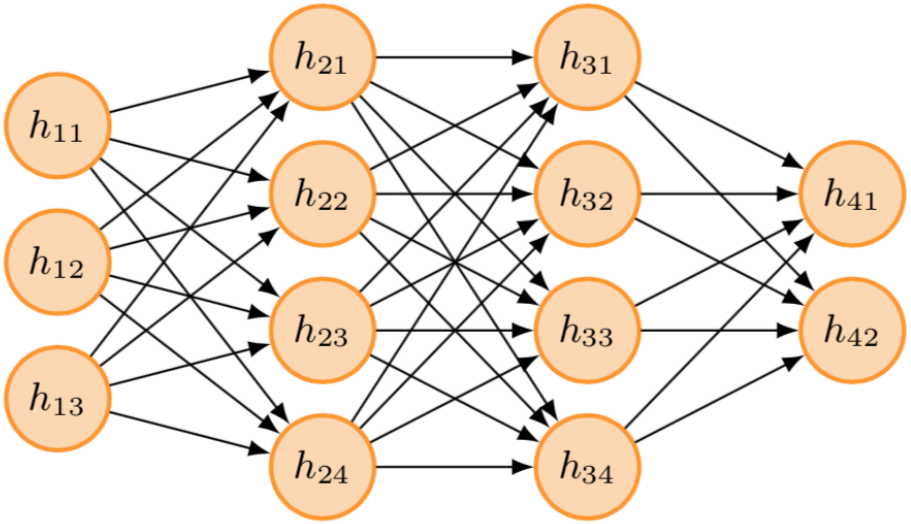
Draw 100 Bernoulli RV's

w.p. p

$$M = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \text{w.p. } p \\ \text{w.p. } 1-p \end{matrix} \qquad h \odot M$$

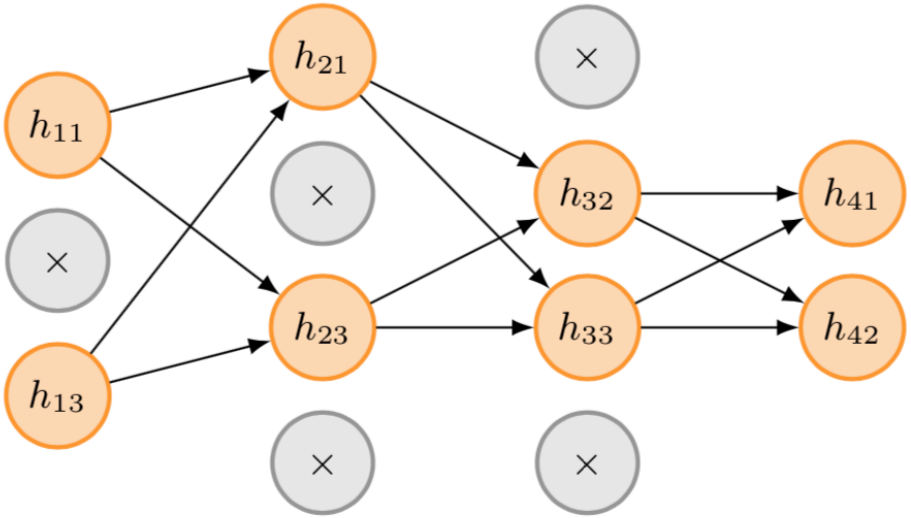
Dropout

Hidden layers of network to be trained



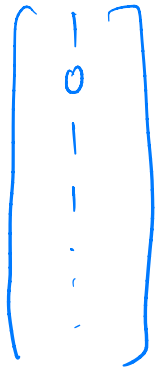
$\mathbb{R}^{13}$

Application of random mask on iteration i



Mask

$M \in \mathbb{R}^{13}$



N units

2^N possible configurations

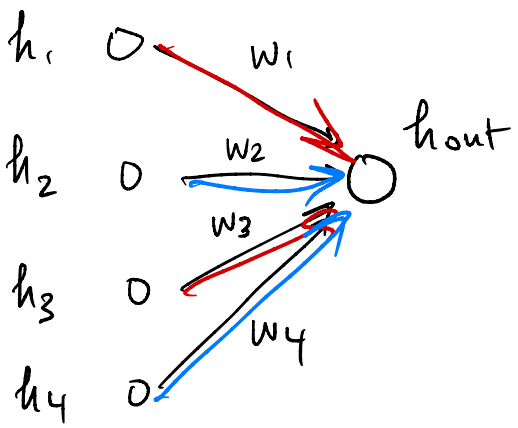
Dropout

Dropout in code.

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = np.random.rand(*H1.shape) < p # Sample random mask
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = np.random.rand(*H2.shape) < p # Sample random mask
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3
```

Dropout

How about during test time? What configuration do you use?



$p = 0.5$

Training iter 1:

$$M = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow h_{out} = \text{relu}(w_1 h_1 + w_3 h_3)$$

Network 1 p

Training iter 2

$$M = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \Rightarrow h_{out} = \text{relu}(w_2 h_2 + w_4 h_4)$$

Network 2 p

TEST:

$$h_{out} = \text{relu}((w_1 h_1 + w_2 h_2 + w_3 h_3 + w_4 h_4)\, p)$$

Over many iterations, the contribution $w_i h_i$ to h_{out} is actually $p \cdot w_i \cdot h_i$.

Dropout

How about during test time? What configuration do you use?

*We call this approach the **weight scaling inference rule**. There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.*

In this class, instead of scaling the weights, we'll scale the activations.

Dropout

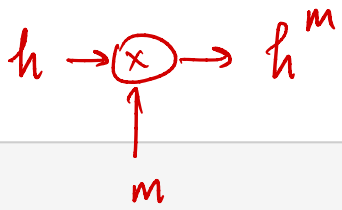
```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def test(X):
    H1 = relu(np.dot(W1, X) + b1) * p
    H2 = relu(np.dot(W1, H1) + b1) * p
    Z = np.dot(W3, H2) + b3
```

Note: an additional pro of dropout is that in testing time, there is no additional complexity. With *m* ensemble models, our test time evaluation would scale *O(m)*.

Inverted dropout

A common way to implement dropout is *inverted dropout* where the scaling by $1/p$ is done in training. This causes the output to have the same expected value as if dropout was never been performed.

Thus, testing looks the same irrespective of if we use dropout or not. See code below:




```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def train_forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = (np.random.rand(*H1.shape) < p) / p # Sample random mask AND normalization by p
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = (np.random.rand(*H2.shape) < p) / p # Sample random mask AND normalization by p
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3

def test(X):
    H1 = relu(np.dot(W1, X) + b1)
    H2 = relu(np.dot(W1, H1) + b1)
    Z = np.dot(W3, H2) + b3
```

Dropout

How is this a good idea?

Goodfellow

1) Dropout approximates bagging, since each mask is like a different model. For a model with N hidden units, there are 2^N different model configurations.

   Each of these configurations must be good at predicting the output.

2) You can think of of dropout as regularizing each hidden unit to work well in many different contexts.

   "masks"

3) Dropout may cause units to encode redundant features (e.g., to detect a cat, there are many things we look for, e.g., it's furry, it has pointy ears, it has a tail, a long body, etc.).

Lecture summary

Here, we've covered tricks that we can do in initialization, regularization, and data augmentation to improve the performance of neural networks.

But what about the optimizer, stochastic gradient descent? Can we improve this for deep learning?

That's the topic of our next lecture.