# Announcements, 2018-02-05

- HW #3 due this Weds, 11:59pm to Gradescope.
  - HW #3, question #1, no 4D tensor derivatives.
  - HW #3, question #2 cancelled.
- HW #3, coding question FC_nets:

## Solver

We will now use the cs231n Solver class to train these networks. Familiarize yourself with the API in `cs231n/solver.py`. After you have done so, declare an instance of a TwoLayerNet with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least ~~50%~~ 40%.

In [ ]:
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 40%.  We won't have you optimize this further
#   since you did it in the previous notebook.
# ================================================================ #

pass

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

# RECAP: Xavier initialization

## Xavier initialization

One initialization is from Glorot and Bengio, 2011, referred to commonly as the Xavier initialization. It intuitively argues that the variance of the units across all layers ought be the same, and that the same holds true for the backpropagated gradients.

$W \approx$ small $\Rightarrow$ activations go to 0.

$W \approx$ large $\Rightarrow$ activations grow unbounded

# RECAP: Xavier initialization

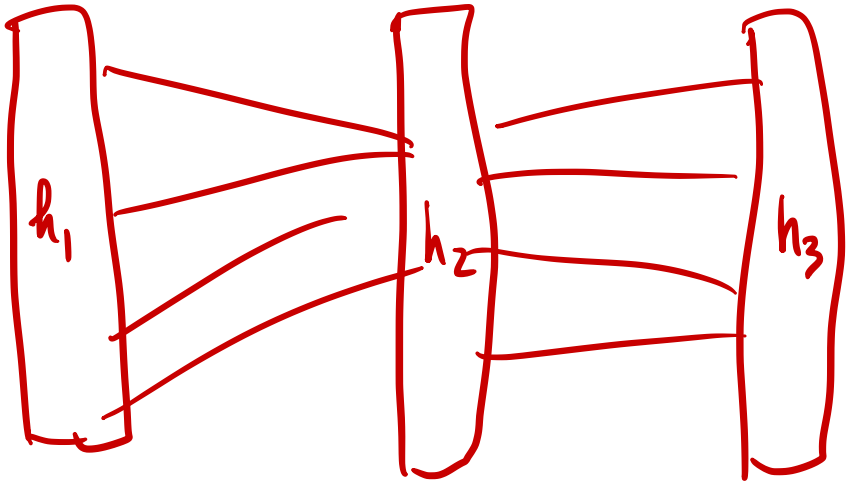
For simplicity, we assume the input has equal variance across all dimensions. Then, each unit in each layer ought to have the same statistics. For simplicity we'll denote $h_i$ to denote a unit in the $i$th layer, but the variances ought be the same for all units in this layer.

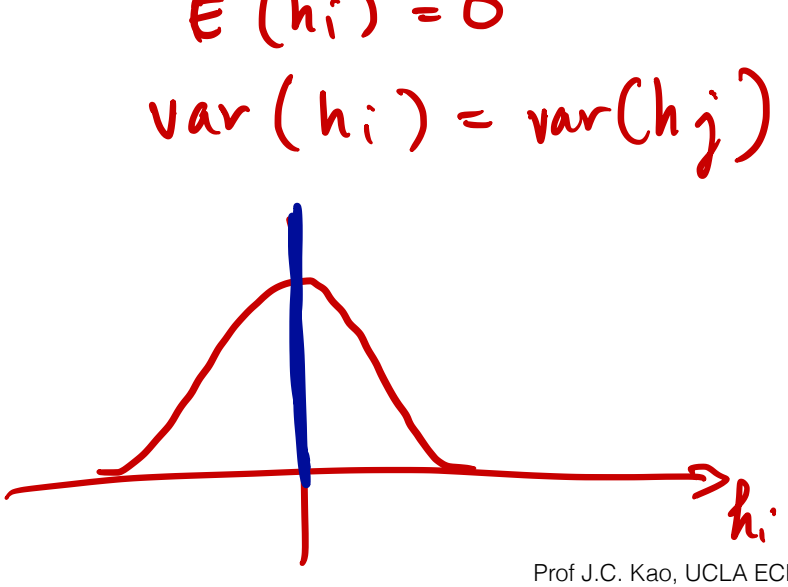
Concretely, the heuristics mean that

$$\text{var}(h_i) = \text{var}(h_j)$$

and

$$\nabla_{h_i} J = \nabla_{h_j} J$$





## Xavier initialization (cont).

If the units are linear, then

$$h_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} h_{i-1,j}$$

Further, if the $w_{ij}$ and $h_{j-1}$ are independent, and all the units in the $(i-1)$th layer have the same statistics, then using the fact that

$$\begin{aligned} \text{var}(wh) &= \mathbb{E}^2(w)\text{var}(h) + \mathbb{E}^2(h)\text{var}(w) + \text{var}(w)\text{var}(h) \\ &= \text{var}(w)\text{var}(h) \qquad \text{if } \mathbb{E}(w) = \mathbb{E}(h) = 0 \end{aligned}$$

then,

$$\text{var}(h_i) = \text{var}(h_{i-1}) \cdot \sum_{i=1}^{n_{\text{in}}} \text{var}(w_{ij})$$

# Xavier initialization

Now if the weights are identically distributed, then we get that for the unit activation variances to be equal,

$$\text{var}(w_{ij}) = \frac{1}{n_{\text{in}}}$$

for each connection $j$ in layer $i$. The same argument can be made for the backpropagated gradients to argue that:

$$\text{var}(w_{ij}) = \frac{1}{n_{\text{out}}}$$



## Xavier initialization (cont).

To incorporate both of these constraints, we can average the number of units together, so that

$$\text{var}(w_{ij}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

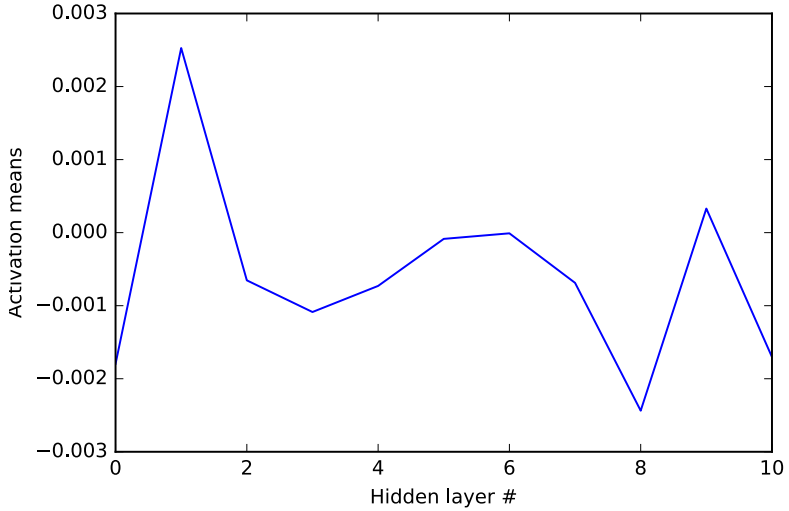
Hence, we can initialize each weight in layer $i$ to be drawn from:

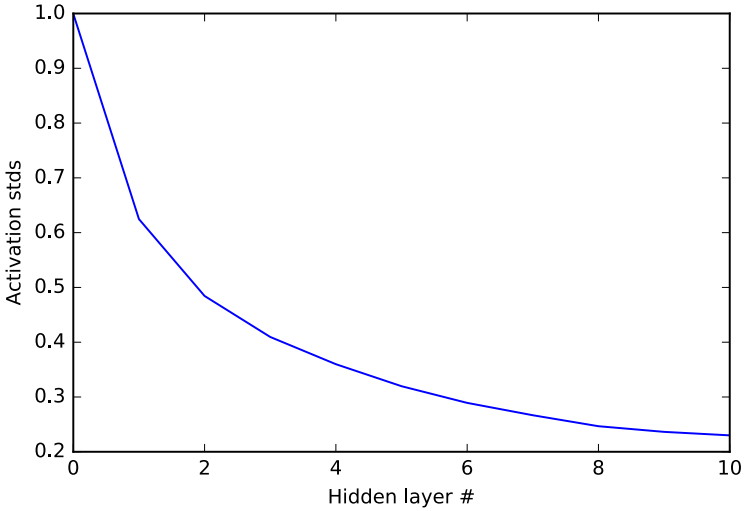
$$\mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

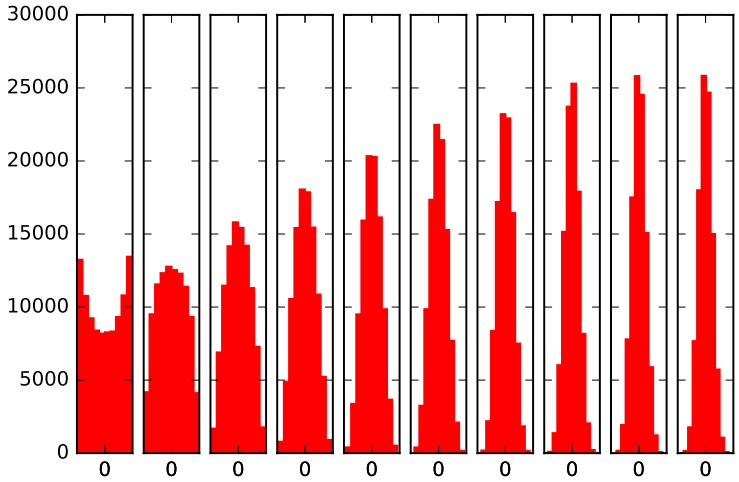
# Xavier initialization with tanh

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / (np.sqrt(100 + 100))
    Z = np.dot(W, H)
    H = np.tanh(Z)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

# Xavier initialization with tanh



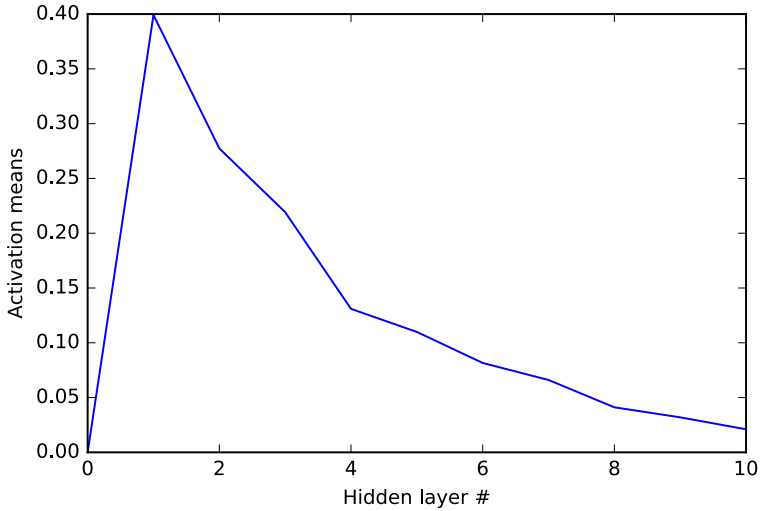
# Xavier initialization

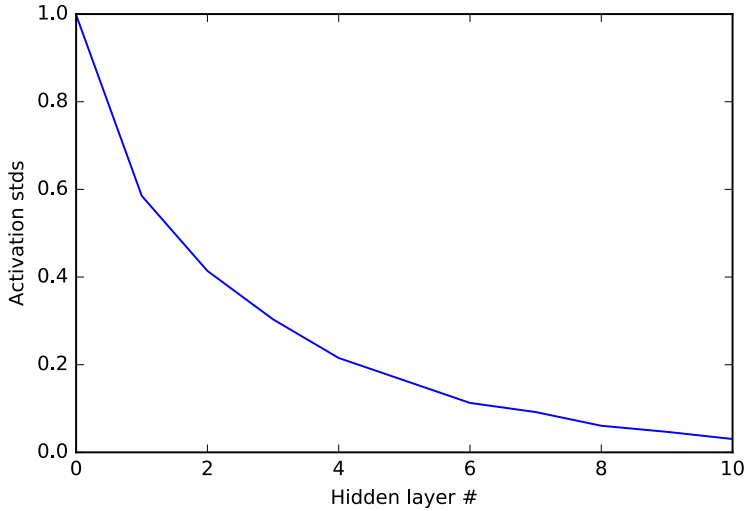
A few notes:

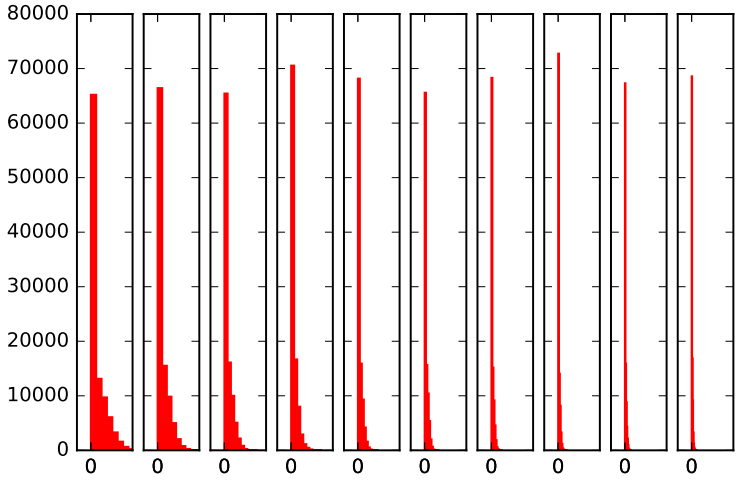
- If you use Caffe (and / or talk to other people), the Xavier initialization sets the variance to $1/n_{\text{in}}$. This is equivalent to the above form if $n_{\text{in}} = n_{\text{out}}$.
- However, the Xavier initialization (either one) typically leads to dying $\text{ReLU}$ units, though it is fine with $\tanh$ units.
- He et al., 2015 suggest the normalizer $2/n_{\text{in}}$ when considering ReLU units. If linear activations prior to the ReLU are equally likely to be positive or negative, $\text{ReLU}$ kills half of the units, and so the variance decreases by half. This motivates the additional factor of $2$.
- Glorot and Bengio, 2015, ultimately suggest the weights be drawn from:

$$U\left(-\frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}, \frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}\right)$$

# Xavier initialization with ReLU



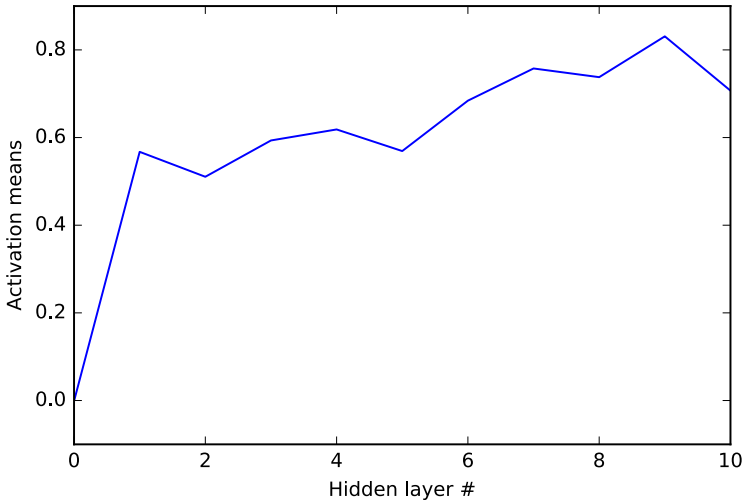



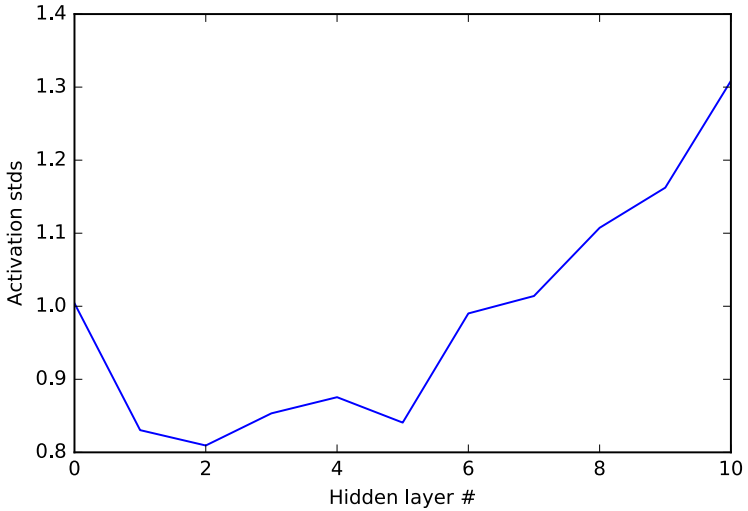



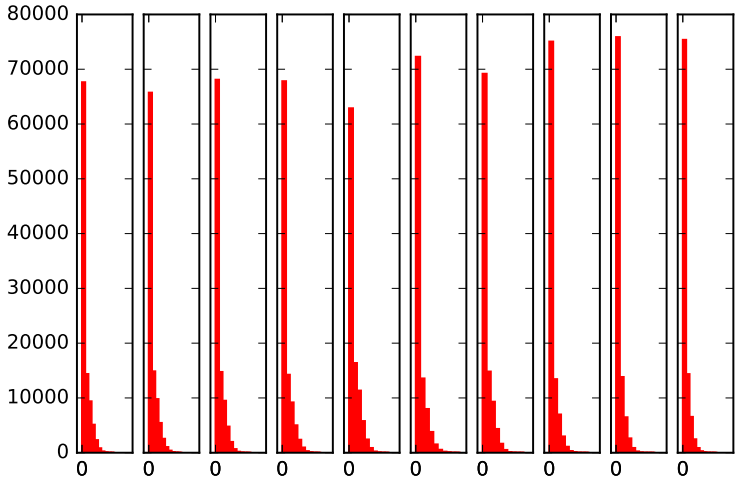
# He (MSRA) initialization with ReLU

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / np.sqrt(100)
    Z = np.dot(W, H)
    H = Z * (Z>0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

# He (MSRA) initialization with ReLU



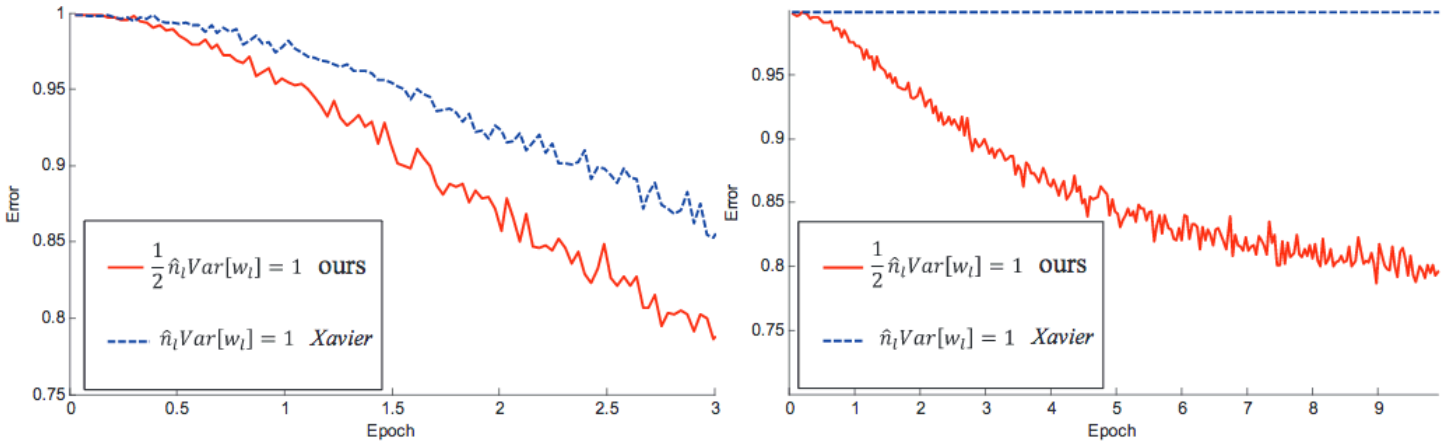



# Initialization take home points

Initialization is a **very** important aspect of training neural networks and remains an active area of research.

A factor of two in the initialization can be the difference between the network learning well or not learning at all.




He et al., 2015


Sussillo and Abbott, "Random walk initialization for training very deep feedforward networks," 2014.
He et al., "Delving deep into rectifiers; surpassing human-level performance on ImageNet classification," 2015.
Mishkin and Matas, "All you need is a good init," 2015.

# Initialization take home points

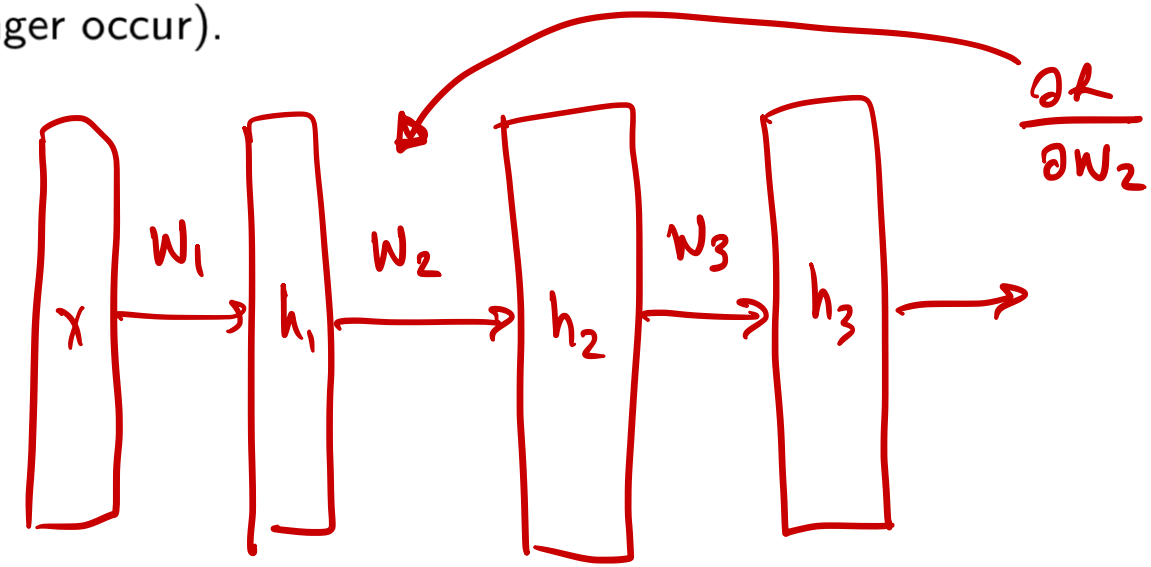
E.g., Mishkin and Matas, “All you need is a good init,” 2015.

| Init method | maxout | ReLU | VLReLU | tanh | Sigmoid |
|---|---|---|---|---|---|
| LSUV | **93.94** | **92.11** | 92.97 | 89.28 | n/c |
| OrthoNorm | 93.78 | 91.74 | 92.40 | 89.48 | n/c |
| OrthoNorm-MSRA scaled | – | 91.93 | **93.09** | – | n/c |
| Xavier | 91.75 | 90.63 | 92.27 | **89.82** | n/c |
| MSRA | n/c† | 90.91 | 92.43 | 89.54 | n/c |

# Avoiding saturation, high variance activations, etc.

An obstacle to standard training is that the distribution of the inputs to each layer changes as learning occurs in previous layers. As a result, the unit activations can be very variable. Another consideration is that when we do gradient descent, we're calculating how to update each paramer *assuming the other layers don't change*. But these layers may change drastically. Ultimately, these cause:

- Learning rates to be smaller (than if the distributions were not so variable).
- Networks to be more sensitive to initializations.
- Difficulties in training networks that saturate (where learning will no longer occur).



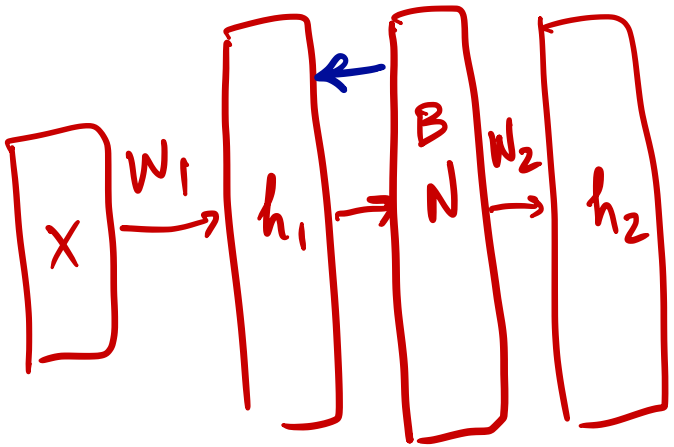
# Batch normalization

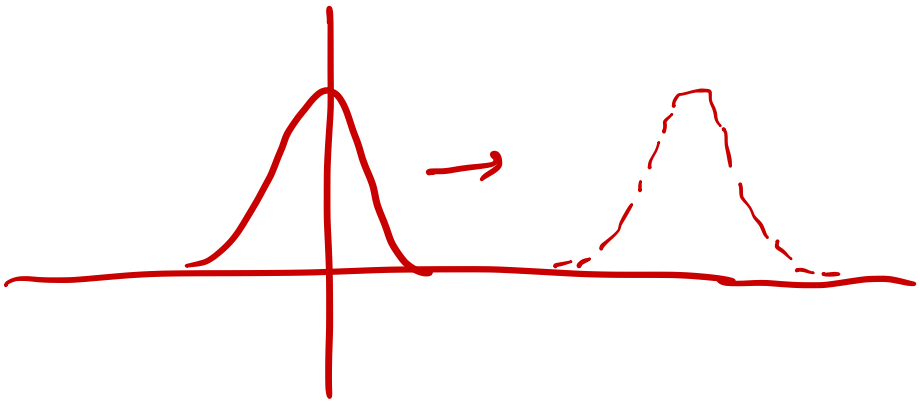
Ioffe & Szegedy 2015

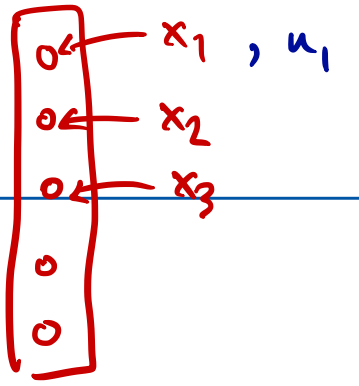
The idea of batch-normalization is to make the output of each layer have unit Gaussian statistics. Learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer. This makes learning more straightforward.

$h_1 \rightarrow Ex = 0$

$var(h_1) \simeq 1$






## Batch normalization (cont.)

(Ioffe and Szegedy, 2015), introduced batch normalization.

- Normalize the unit activations:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

with

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

and $\varepsilon$ small.

- Scale and shift the normalized activations:

$$y_i^{(j)} = \gamma_i \hat{x}_i^{(j)} + \beta_i$$

$x_i$ : refers to the $i^{th}$ neuron activation in a given layer

$x_i^{(j)}$

Affine → BatchNorm → Relu

Importantly, the normalization and scale / shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation.



## Batch normalization (cont.)

A few notes on batch normalization's implementation:

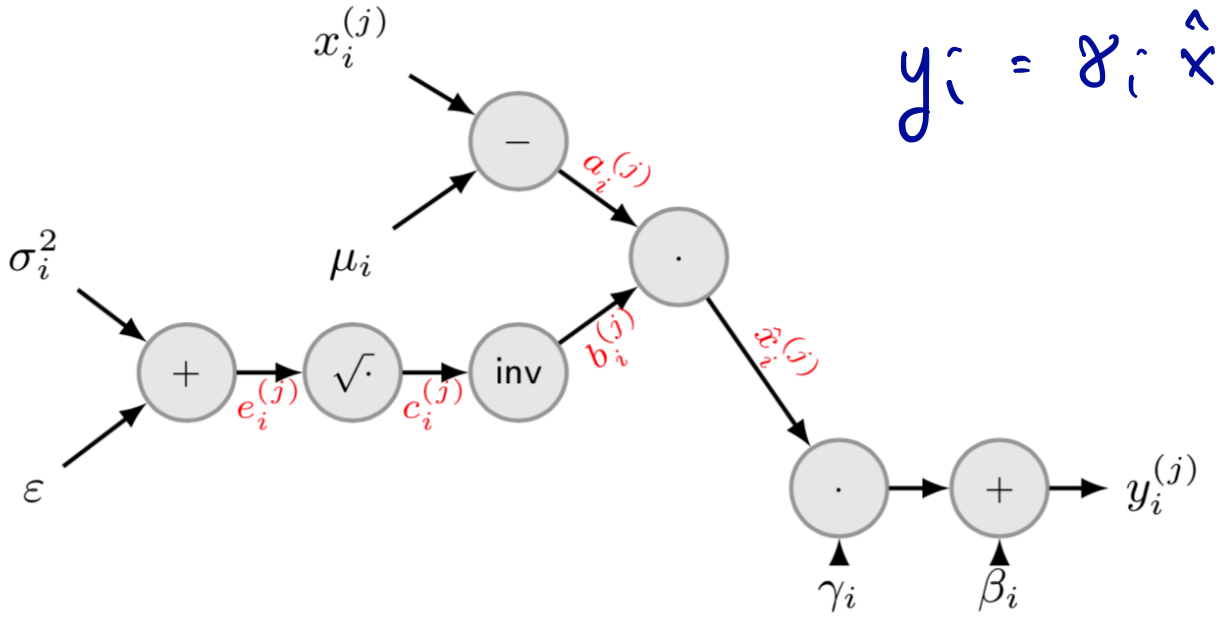
- The reason the scaling is on a per unit basis is primarily computational efficiency. It is possible to normalize the entire layer via $\Sigma^{-1/2}(\mathbf{x} - \mu)$ where $\mu, \Sigma$ are the mean and covariance of $\mathbf{x}$. However, this requires computation of a covariance matrix, its inverse, and the appropriate terms for backpropagation (including computation of the Jacobian of this transform).
- The scale and shift layer is inserted in case it is better that the activations not be zero mean and unit variance. As $\gamma_i$ and $\beta_i$ are parameters, it is possible for the network to rescale the activations. In fact, it could learn $\gamma_i = \sigma_i$ and $\beta_i = \mu_i$ to undo the normalization.

# Batch normalization

## Batch normalization computational graph

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

$$y_i = \gamma_i \hat{x}_i + \beta_i$$




Gradients from backprop on next page.

affine - relu - affine - relu ...

affine - batchnorm - relu - ...

FC_nets

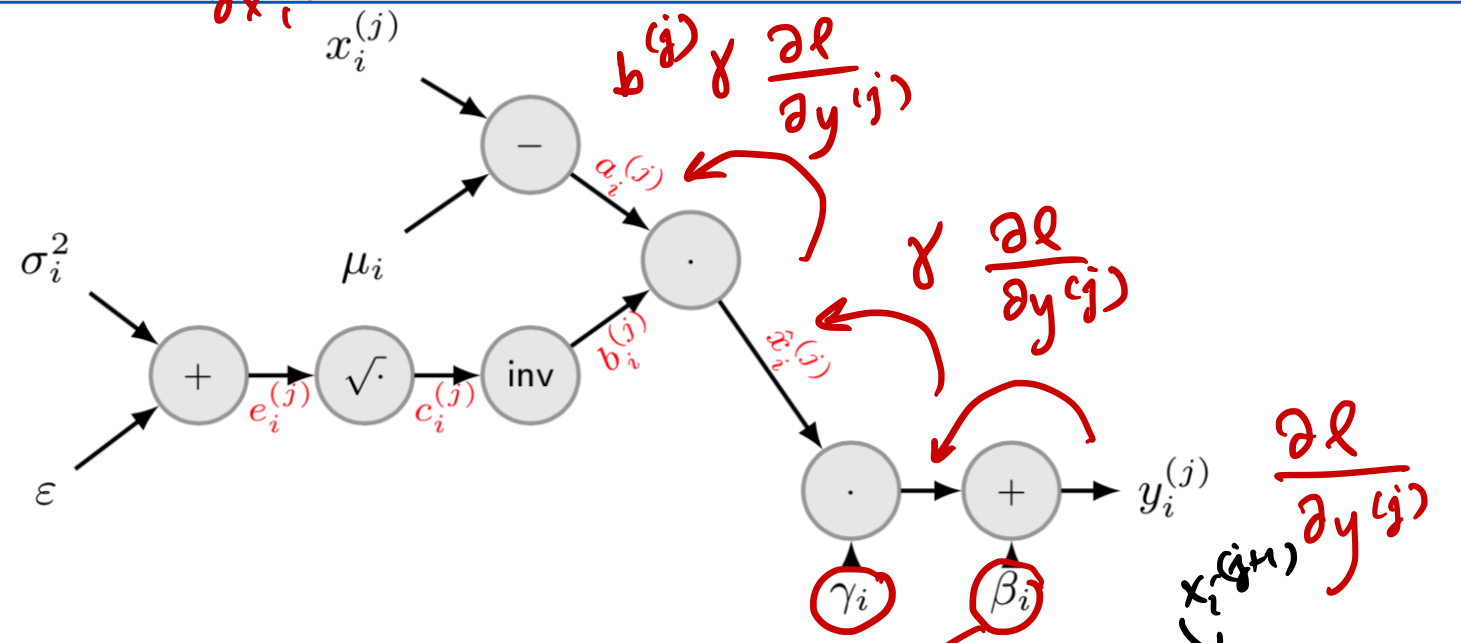
layers.py

affine: $Wx + b$

relu: relu(x)

batchnorm: BN(x)

# Batch normalization



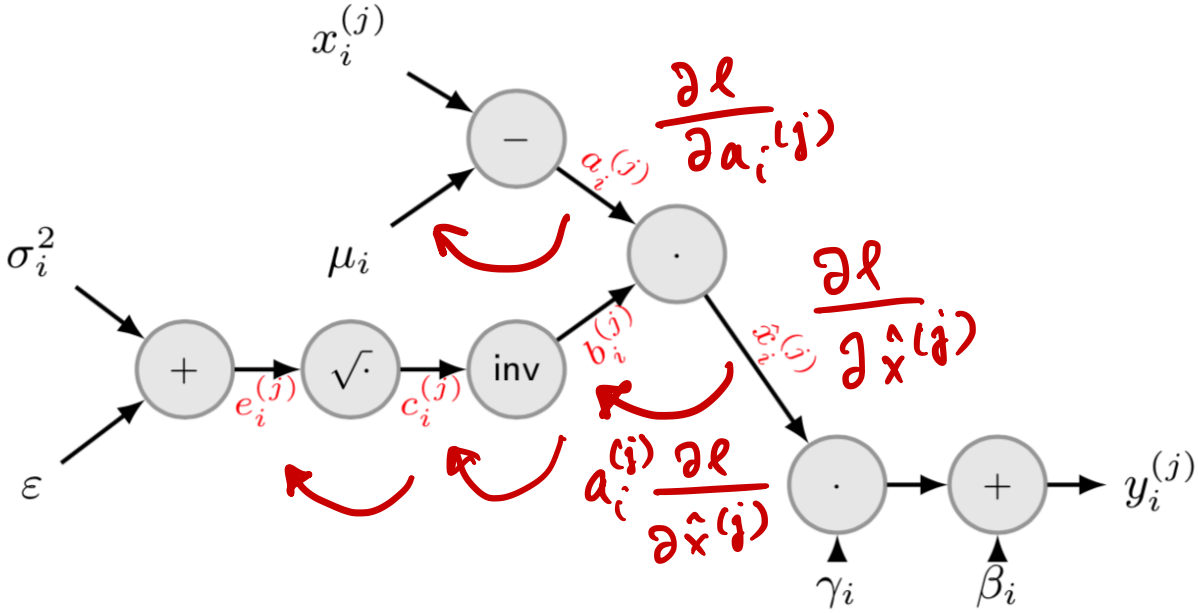

$$\frac{\partial \ell}{\partial \beta} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \hat{x}^{(j)}$$

$$\frac{\partial \ell}{\partial \hat{x}^{(j)}} = \frac{\partial \ell}{\partial y^{(j)}} \gamma$$

$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

# Batch normalization



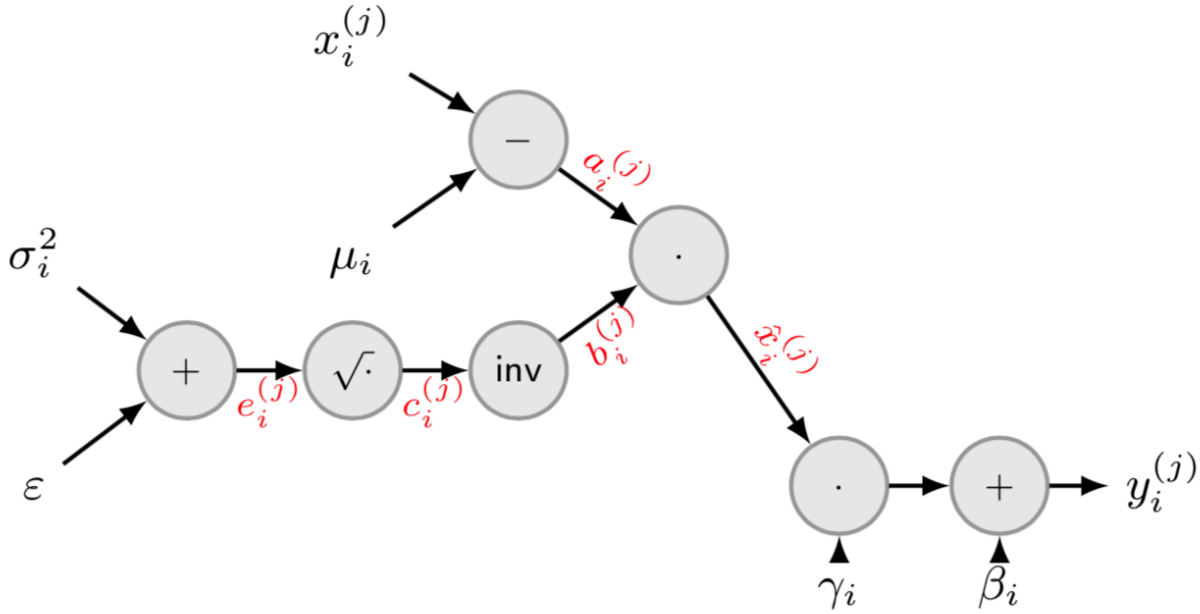

$$\frac{\partial \ell}{\partial \mu} = -\frac{1}{\sqrt{\sigma^2+\epsilon}} \sum_{j=1}^{m} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial b^{(j)}} = (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial c^{(j)}} = -\frac{1}{\sigma^2+\varepsilon}(x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2}\frac{1}{(\sigma^2+\varepsilon)^{3/2}}(x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

# Batch normalization

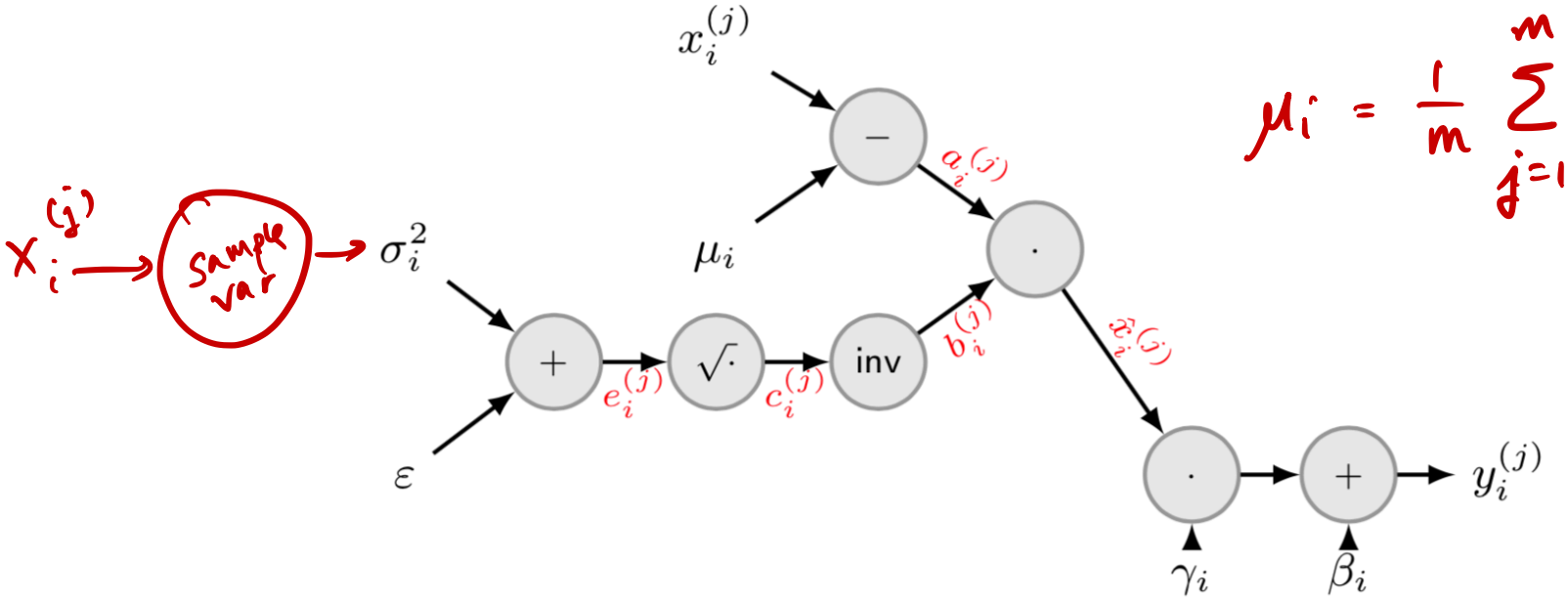



$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \qquad \frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial \sigma^2} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial e^{(j)}}$$

$$= \sum_{j=1}^{m} -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

# Batch normalization




$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}}\frac{\partial \ell}{\partial \hat{x}^{(j)}} \qquad \frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2}\frac{1}{(\sigma^2+\varepsilon)^{3/2}}(x^{(j)}-\mu)\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\sigma^2 = \frac{1}{m}\sum_{j=1}^{m}(x^{(j)}-\mu)^2$$

$$\frac{\partial \ell}{\partial x^{(j)}} = \frac{\partial \ell}{\partial a^{(j)}} + \frac{\partial \sigma^2}{\partial x^{(j)}}\frac{\partial \ell}{\partial \sigma^2} + \frac{\partial \mu}{\partial x^{(j)}}\frac{\partial \ell}{\partial \mu}$$

$$= \frac{1}{\sqrt{\sigma^2+\epsilon}}\frac{\partial \ell}{\partial \hat{x}^{(j)}} + \frac{2(x^{(j)}-\mu)}{m}\frac{\partial \ell}{\partial \sigma^2} + \frac{1}{m}\frac{\partial \ell}{\partial \mu}$$

$$\frac{\partial \sigma^2}{\partial x^{(j)}} = \frac{2}{m}(x^{(j)}-\mu)$$



## Batch normalization layer

The batch normalization layer is typically placed right before the nonlinear activation. Hence, a layer of a neural network may look like:

$$\mathbf{h}_i = f\left(\text{batch-norm}\left(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i\right)\right)$$

batch norm $(h_i)$

# Batch normalization

Empirically, batch normalization:

- Allows higher learning rates to be used.
- Reduces the strong dependence on initialization.


# What is regularization?

# What is regularization?

$w_{ij} \approx b$

$J(W) + \lambda \|W - b\|_F$

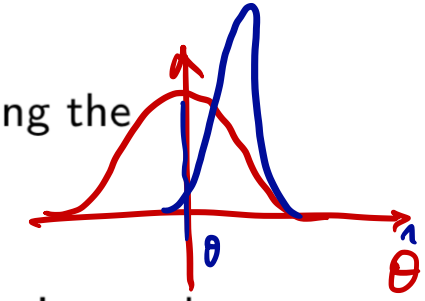
## Regularizations

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.



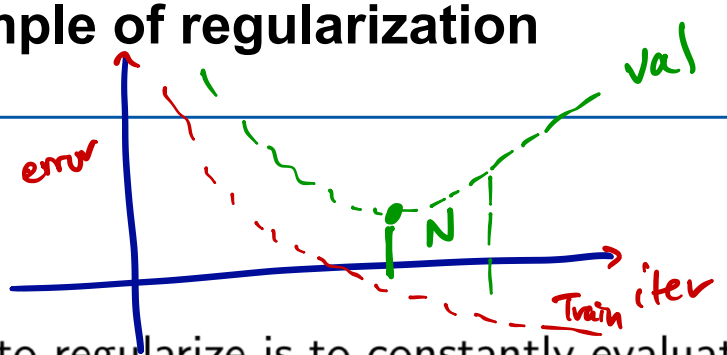
# What is regularization?

## Types of regularization

Regularizations may take on many different types of forms. The following list is not exhaustive, but includes regularizations one may consider.

- It may be appropriate to add a soft constraint on the parameter values in the objective function.
  - To account for prior knowledge (e.g., that the parameters have a bias).
  - To prefer simpler model classes that promote generalization.
  - To make an underdetermined problem determined. (e.g., least squares with indeterminate $\mathbf{X}^T\mathbf{X}$.)
- Dataset augmentation
- Ensemble methods (i.e., essentially combining the output of several models).
- Some training algorithms (e.g., stopping training early, dropout) can be seen as a type of regularization.

# 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.

# Parameter norm penalties

It is common to associate regularization with 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).

## 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:

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

$\Omega(\theta) = \|\theta\|$

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

- $\alpha$ 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 $\alpha$ can strongly affect generalization performance.
- When regularizing parameters, we typically do *not* regularize biases, since they do not introduce substantial variance to the estimator.

# 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:

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

$$\frac{1}{2}\|W\|_F^2$$

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

$$\|W\|_F^2 = \sum_i \sum_j (w_{ij})^2$$

$$\frac{\partial}{\partial W}\frac{1}{2}\|W\|_F^2 = W$$

# 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:

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

$W \leftarrow W - \varepsilon \nabla_W J$

$W \leftarrow W - \varepsilon \nabla_W \tilde{J}$

$= W - \varepsilon(\alpha W + \nabla_W J)$

$= (1 - \varepsilon\alpha) W - \nabla_W J$

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

# 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 $i$th 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.



## 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:

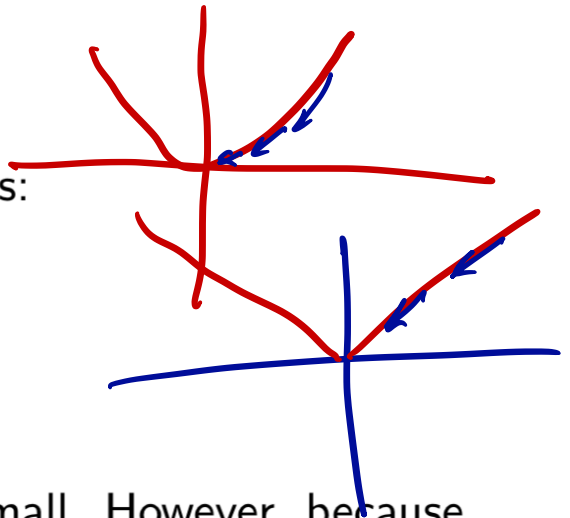
$$\Omega(\theta) = \|\mathbf{w} - \mathbf{b}\|_2$$

- 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:

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

# 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:

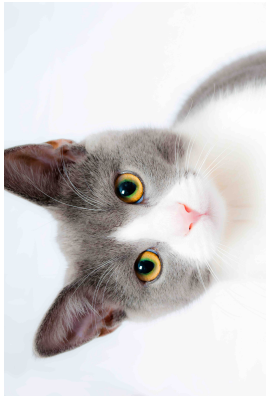
Flipped image:

Cropped image:

Adjust brightness

Lens correction

Rotate



# Dataset augmentation

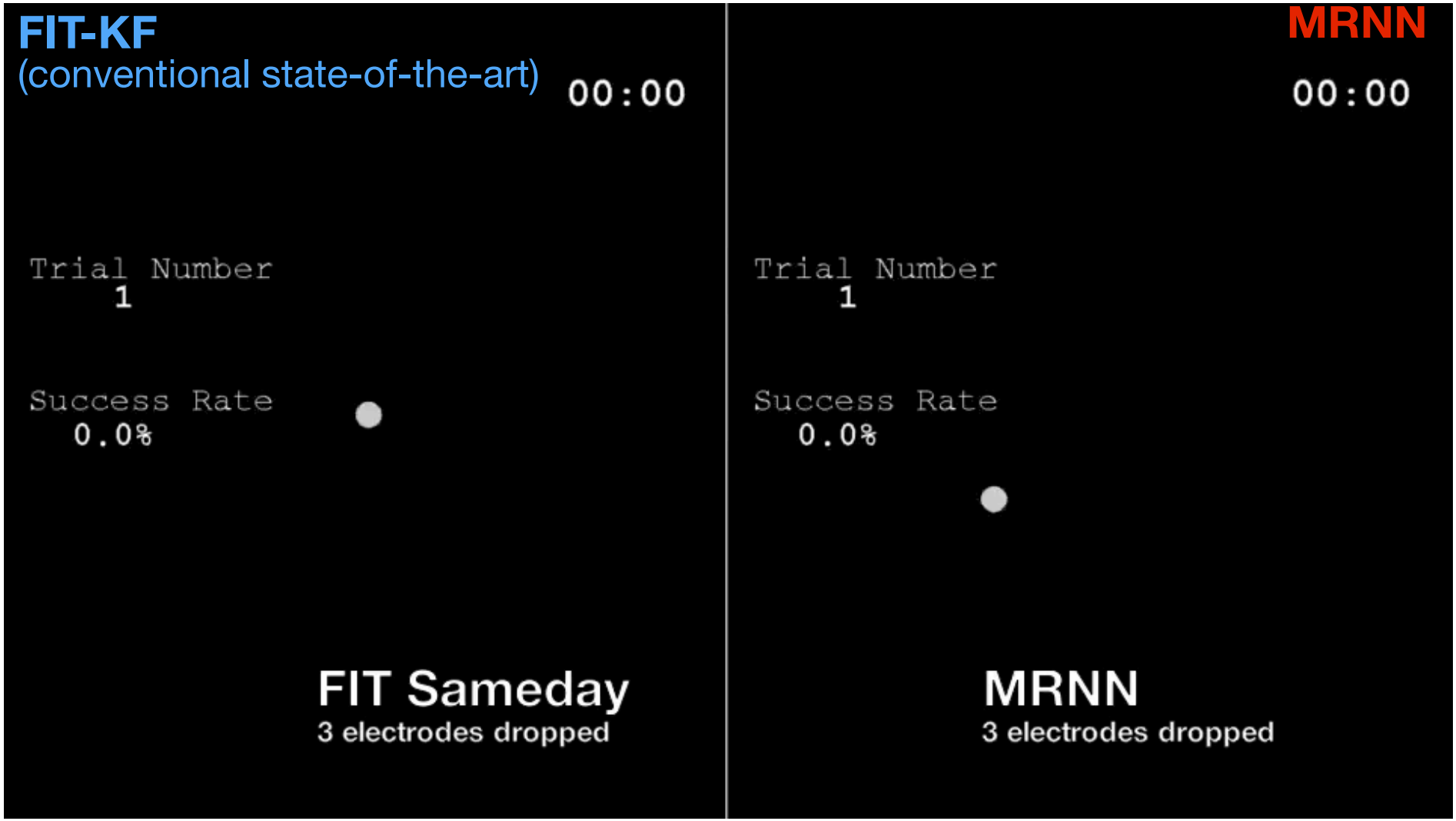
There are various heuristics to keeping the input size the same.

You can sample smaller patches.  When resizing or scaling, you can sample a portion of the image but have the minimum dimension still be larger than the patch size.

You can be creative for dataset augmentation.

# Dataset augmentation

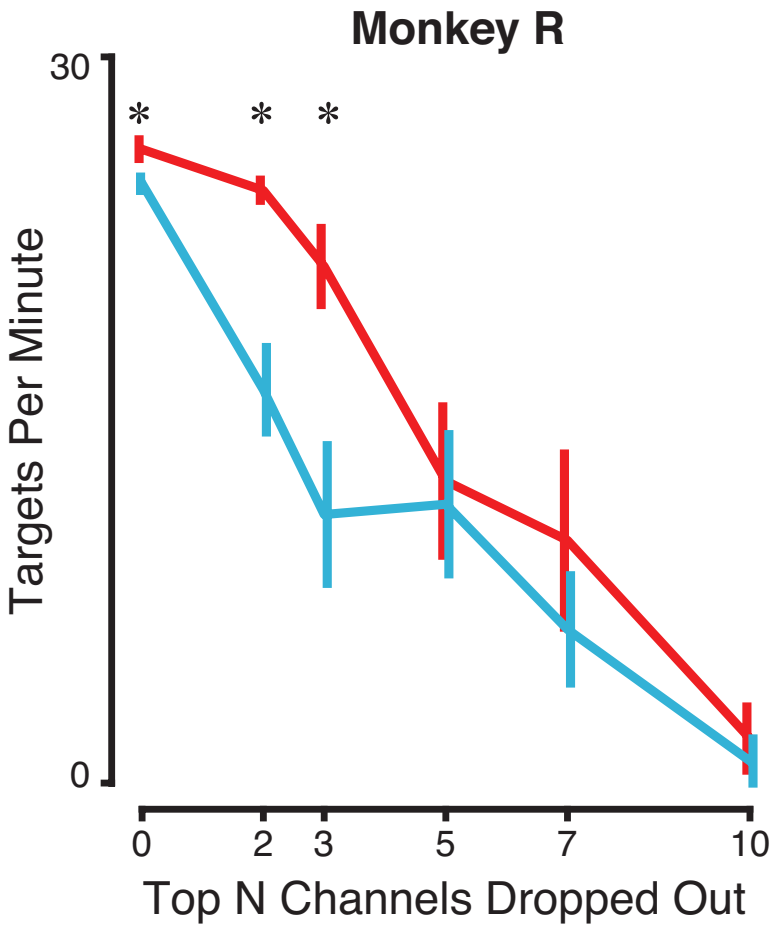
Example from brain-machine interfaces:

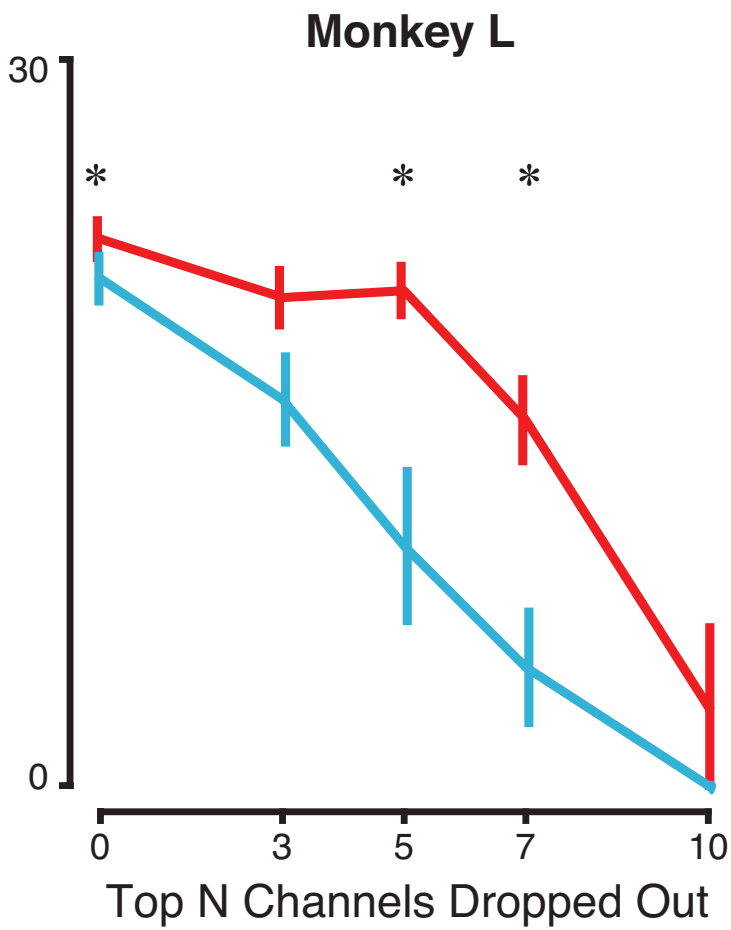



(back-to-back comparisons during the same experiment)

# Dataset augmentation

Example from brain-machine interfaces:



# Dataset augmentation

$y^{(i)} \in \{1, \ldots, 10\}$

w.p. $1-\epsilon$

Other types of dataset augmentation:

- Inject noise into the network
- Label smoothing

1 ... 10

w.p. $1-\epsilon$

| 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 $7 \times 7$ | 21.6% | 5.8 | 4.8 |
| Inception-v2 BN-auxiliary | **21.2%** | **5.6%** | 4.8 |

0.6%

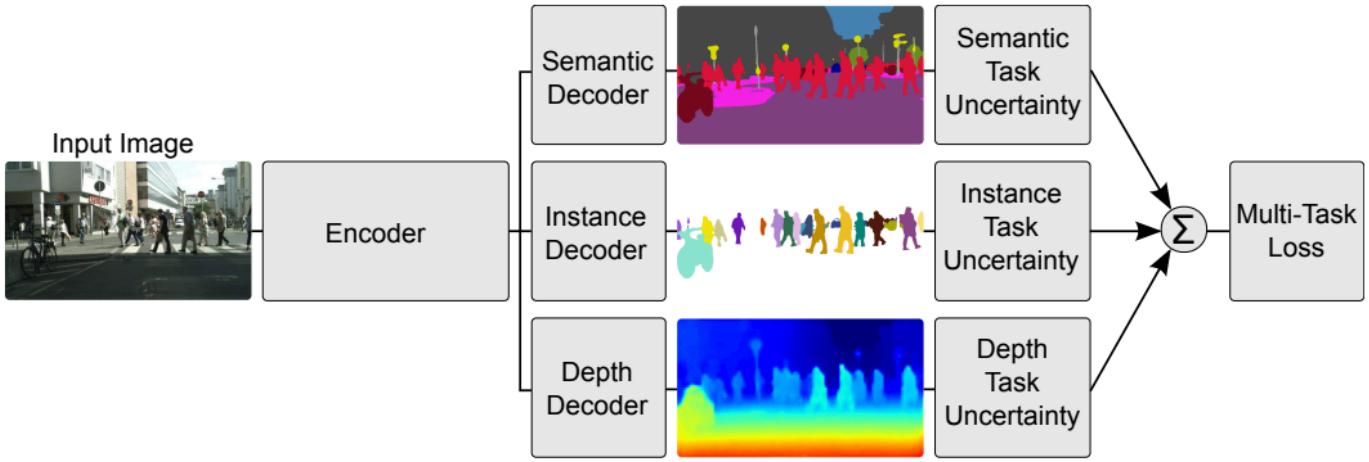
Szegedy et al., arXiv 2015



## 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.

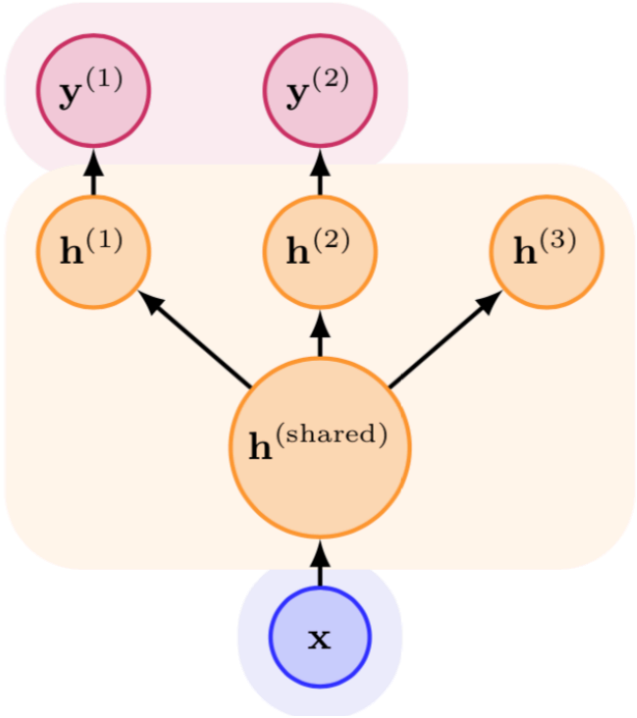
# Multitask learning




Kendall et al., arXiv 2017

# Multitask learning

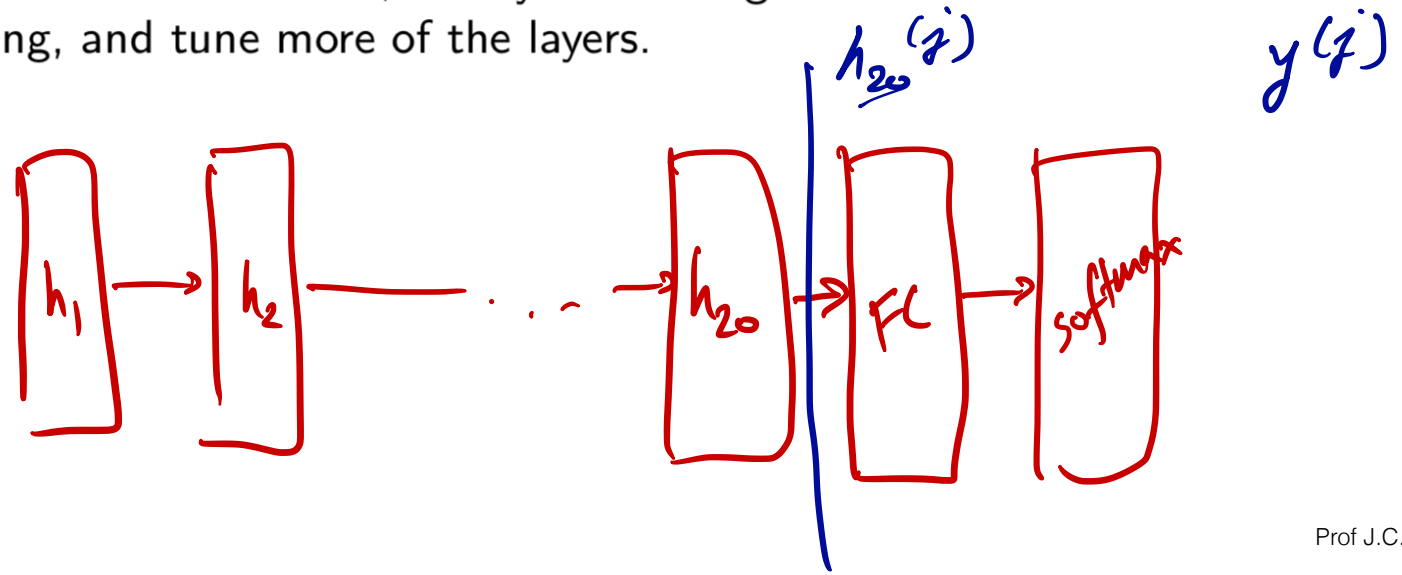
- The entire model need not be shared across different tasks.
- Here, $\mathbf{h}^{\text{shared}}$ captures common features that are then used by task-specific layers to predict $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$.
- $\mathbf{h}^{(3)}$ could represent a feature for unsupervised learning.



## 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 $\epsilon_i$ 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,

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

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

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

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