Training neural networks

- Weight initialization
- Pre-processing
- Batch normalization
- Regularizations
  - L^2 and L^1 regularization
  - Dataset augmentation
  - Multitask and transfer learning
  - Ensemble methods
  - Dropout

Weight initialization

Weight initialization in neural networks can be very important for converging to a proper solution. It can be thought of as “seeding” the neural network at a good starting point for gradient descent.

- Some initializations are obviously bad. For example, for a neural network with an initializtaion of $\mathbf{W} = 0$ and $\mathbf{b} = 0$ and with an activation function where $f(0) = 0$, there would not be any any learning.
- An initialization of $\mathbf{W} = 0$ will result in all the neuron activations being the same, and thus all the gradients being the same, and hence all parameters receiving the same update. The fundamental problem here is a symmetry in the parameters.
- Initialization to small random weights will have activations that decay to zero in deeper layers. This also means very little learning will happen in deeper layers, as the gradient is multiplied by the small activations.
- Similarly, initialization to large random weights will have activations that saturate (for saturating nonlinearities, e.g., $\tanh$) or grow unbounded (e.g., for ReLU).
- The weights must be initialized to a good range of values to avoid zero, saturating or exploding activations.

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.

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 ith 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

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

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_{i-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,

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

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:

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

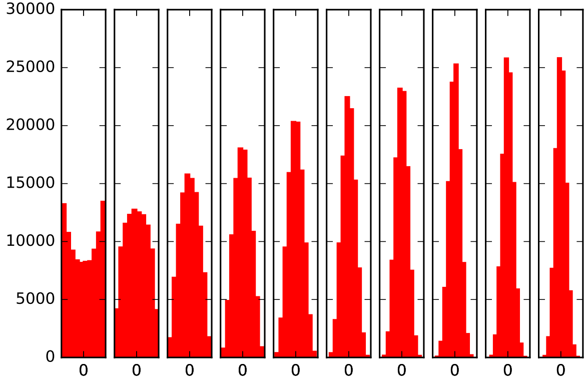
A few notes:

- 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 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, ReLU kills half of the units, and so the variance decreases by half. This motivates the additional factor of 2.
- Glorot and Bengio, 2011, ultimately suggest the weights be drawn from:

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

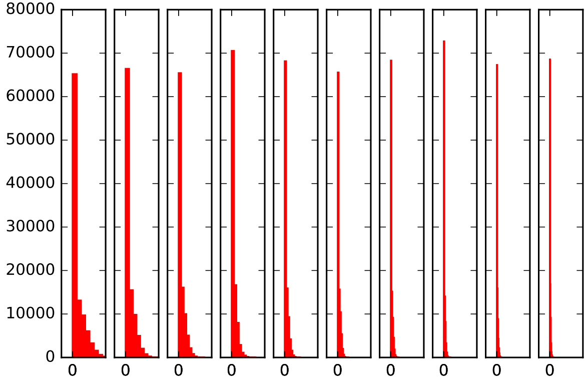
Example of Xavier initialization with tanh

The following are the tanh unit activations of a 10 layer network where each hidden layer has 100 units. Each plot corresponds to a layer in the network.



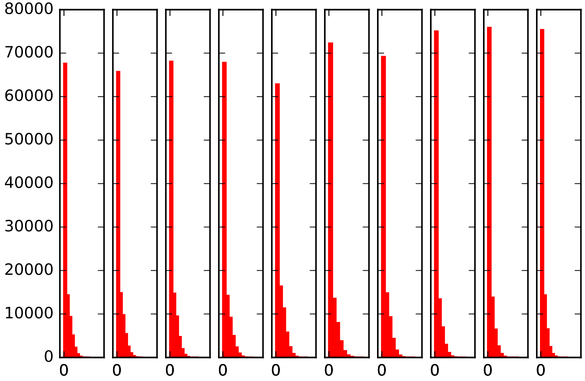
Example of Xavier initialization with ReLU

The following are the ReLU unit activations of a 10 layer network where each hidden layer has 100 units. Each plot corresponds to a layer in the network.



Example of He initialization with ReLU

The following are the ReLU unit activations of a 10 layer network where each hidden layer has 100 units. Each plot corresponds to a layer in the network.



Preprocessing of inputs

In the Neural Networks lecture, we saw that a con of the sigmoid unit was that it was not mean-centered. This results in units that are all positive, and thus weight gradients that are all positive or negative.

For the same reason, it is recommended to zero mean the input data. Note that if you zero mean the data, the mean is calculated from the training set. This mean is then applied to the validation and testing data.

Batch normalization

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).

The idea of batch-normalization is to make the output of each layer have mean zero and standard deviation 1 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.

Batch normalization (cont.)

A naive way to perform such a normalization is to, at the output of each layer, intervene and make the data zero mean and unit variance. However, this does not work well because the learning algorithm may propose changing the mean and variance of the layer, and this normalization undoes that learning.

Another way may be to add penalties to the cost function (regularization, to be described later this lecture) to normalize the activations. However, this normalization is imperfect.

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 ε small.

- Scale and shift the normalized activations:

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

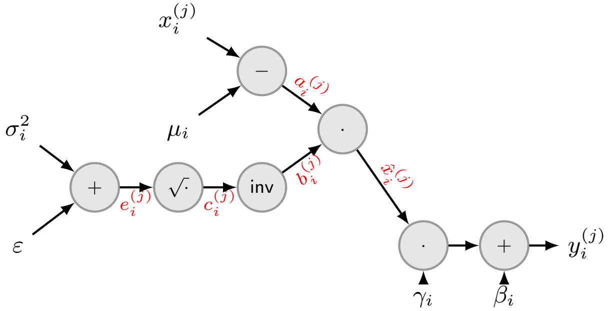
Importantly, the normalization and scale / shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation.

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 μ, Σ 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 γ_i and β_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 computational graph



Gradients from backprop on next page.

Batch normalization computational graph (cont.)

Gradients, given $\frac{\partial \ell}{\partial y}$ (for notational convenience, we will drop the subscripts $\cdot_i$):

$$\begin{aligned}
\frac{\partial \ell}{\partial \beta} &= \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \\
\frac{\partial \ell}{\partial \gamma} &= \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \hat{x}^{(j)} \\
\frac{\partial \ell}{\partial \hat{x}^{(j)}} &= \frac{\partial \ell}{\partial y^{(j)}} \gamma \\
\frac{\partial \ell}{\partial a^{(j)}} &= \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \\
\frac{\partial \ell}{\partial \mu} &= -\frac{1}{\sqrt{\sigma^2+\epsilon}} \sum_{j=1}^{m} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \\
\frac{\partial \ell}{\partial b^{(j)}} &= (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}} \\
\frac{\partial \ell}{\partial c^{(j)}} &= -\frac{1}{\sigma^2+\varepsilon} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}} \\
\frac{\partial \ell}{\partial e^{(j)}} &= -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}
\end{aligned}$$

Batch normalization computational graph (cont.)

$$\begin{aligned}
\frac{\partial \ell}{\partial \sigma^2} &= \sum_{j=1}^{m} \frac{\partial \ell}{\partial e^{(j)}} \\
&= \sum_{j=1}^{m} -\frac{1}{2} \frac{1}{(\sigma^2 + \varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}} \\
\frac{\partial \ell}{\partial x^{(j)}} &= \frac{\partial \ell}{\partial a^{(j)}} + \frac{\partial \sigma^2}{\partial x^{(j)}} \frac{\partial \ell}{\partial \sigma^2} + \frac{\partial \mu}{\partial x^{(j)}} \frac{\partial \ell}{\partial \mu} \\
&= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} + \frac{2(x^{(j)} - \mu)}{m} \frac{\partial \ell}{\partial \sigma^2} + \frac{1}{m} \frac{\partial \ell}{\partial \mu}
\end{aligned}$$

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:

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

Note, this may not be optimal, and people have reported better performance with batch normalization after ReLU.

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.

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: 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.

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:

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

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.
- When regularizing parameters, we typically do *not* regularize biases, since they do not introduce substantial variance to the estimator.

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:

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

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

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.

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.

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:

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

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 $\mathrm{sign}(\mathbf{w})$, 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.,

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

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

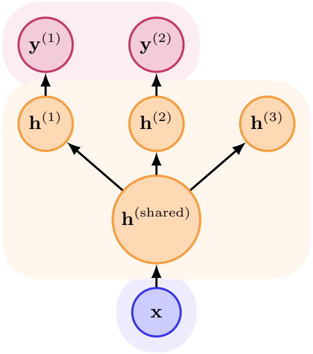
Neural networks generalize better when more training data is available. One way to increase the dataset size is to generate fake data (with appropriate statistics) and add it to the training set.

- This is particularly useful for image and object recognition. The dataset augmentation could be translating, rotating, or scaling the input images. This often results in better generalization in these tasks.
- In brain-machine interface training, augmenting the data by perturbing it intentionally resulted in better robustness.
- We may also consider the injection of input noise as dataset augmentation.
- Yet another type of noise injection is to use *label smoothing* where the outputs are correct with probability $1 - \epsilon$. This accounts for the fact that there may be noise in the labels.

Multitask learning

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

- 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

Ensemble methods train several different models separately. In testing, ensemble methods average the output of these models.

- 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,

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

- 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.

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.
- One apprpoach around this is to take snapshots of the model at different local minima in one training procedure, e.g., Snapshot Ensembling (Huang, Li et al., ICLR 2017).

However, the best way to take advantage of the ensemble methods is to use dropout.

Dropout

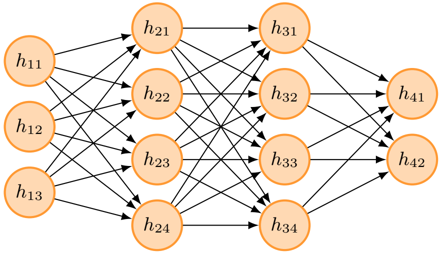
Dropout is a computationally inexpensive yet effective method for generalization. It can be viewed as approximating the bagging procedure for exponentially many models, while only optimizing a single set of parameters. The following steps describe the dropout regularizer:

- 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. This maintains the expected value of the output.

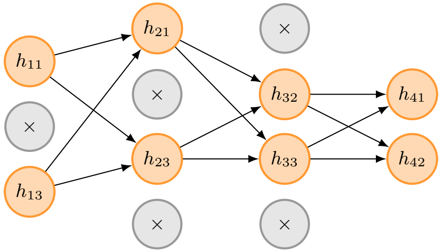
  But note: it is common to scale the weights by $1/p$ in the training phase so that the output at prediction does not have to be scaled. (The expected value of the output after this scaling is the same as if dropout had not been performed.)

Visualizing dropout

Hidden layers of network to be trained



Application of random mask on iteration i



Dropout (cont.)

A few notes:

- Conceptually, we think of dropout as regularizing each hidden unit to work in many different contexts.
- Dropout may be more sophisticated than e.g., simply adding noise. For example, when features are destroyed, dropout may redundantly encode features.
- Doing prediction in the trained model has the same cost as if we had not performed dropout. It does not scale with the number of networks.
- A con of dropout is that, by viewing it as regularization, we see that we may need a larger model. Increasing model size will increase training time.
- Dropout is not as effective when few training examples are available.
- The mask can be analog; it need not be binary.