CS 453X: Class 22

Jacob Whitehill

L_1, L_2 Regularization

Regularization

- **Regularization** is any means to help a machine learning model to generalize better to data not used for training.

- Regularization is particularly important with neural networks since they are so powerful and therefore prone to overfitting.

L_2 regularization in NNs

- To prevent the weight matrices from growing too big, we can apply an L_2 regularization term to each matrix by augmenting the cross-entropy loss:

$$f_{\mathrm{CE}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{10} \mathbf{y}_k^{(i)} \log \hat{\mathbf{y}}_k^{(i)} + \frac{1}{2}\|\mathbf{W}^{(1)}\|_{\mathrm{Fr}}^2 + \frac{1}{2}\|\mathbf{W}^{(2)}\|_{\mathrm{Fr}}^2$$

- Here, $|\mathbf{W}|_{\mathrm{Fr}}^2$ means the squared **Frobenius norm** of **W**.
    - It's just the sum of squares of all the elements of **W**.

L_2 regularization in NNs

- This results in a modified gradient for each weight matrix:

$$
\begin{aligned}
\nabla_{\mathbf{W}^{(2)}} f_{\mathrm{CE}} &= (\hat{\mathbf{y}} - \mathbf{y}){\mathbf{h}^{(1)}}^{\top} + \mathbf{W}^{(2)} \\
\nabla_{\mathbf{W}^{(1)}} f_{\mathrm{CE}} &= \mathbf{g}\mathbf{x}^{\top} + \mathbf{W}^{(1)}
\end{aligned}
$$

L_1 regularization in NNs

- Alternatively, we can use L_1 regularization, which encourages entries of the weight matrix to be exactly 0:

$$f_{\text{CE}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{10} \mathbf{y}_k^{(i)} \log \hat{\mathbf{y}}_k^{(i)} + |\mathbf{W}^{(1)}| + |\mathbf{W}^{(2)}|$$

- Here, $|\mathbf{W}|$ means the sum of the absolute values of each element of $\mathbf{W}$.

L_1 regularization in NNs

- This results in the following gradient terms:

$$
\begin{aligned}
\nabla_{\mathbf{W}^{(2)}} f_{\mathrm{CE}} &= (\hat{\mathbf{y}} - \mathbf{y}){\mathbf{h}^{(1)}}^{\top} + \operatorname{sign}\left(\mathbf{W}^{(2)}\right) \\
\nabla_{\mathbf{W}^{(1)}} f_{\mathrm{CE}} &= \mathbf{g}\mathbf{x}^{\top} + \operatorname{sign}\left(\mathbf{W}^{(1)}\right)
\end{aligned}
$$

$L_1 + L_2$ regularization

- We can also combine both kinds of regularization with different strengths for each weight matrix:

$$\begin{aligned}
\nabla_{\mathbf{W}^{(2)}} f_{\mathrm{CE}} &= (\hat{\mathbf{y}} - \mathbf{y}){\mathbf{h}^{(1)}}^{\top} + \alpha^{(2)}\mathbf{W}^{(2)} + \beta^{(2)}\mathrm{sign}\left(\mathbf{W}^{(2)}\right) \\
\nabla_{\mathbf{W}^{(1)}} f_{\mathrm{CE}} &= \mathbf{g}\mathbf{x}^{\top} + \alpha^{(1)}\mathbf{W}^{(1)} + \beta^{(1)}\mathrm{sign}\left(\mathbf{W}^{(1)}\right)
\end{aligned}$$

- This results in several hyperparameters, all of which should be optimized on a separate validation set (*not* the test set).

Dropout

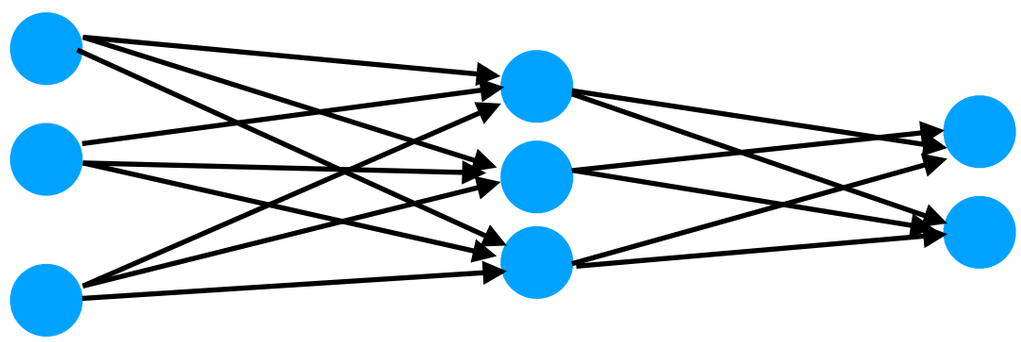
Dropout

- One of the most recently discovered regularization methods is **dropout**, whereby a random set of neurons is removed from the network for each gradient update.

- Surprisingly, this simple method can both help the network to reach a better local minimum *and* prevent it from overfitting.

Dropout

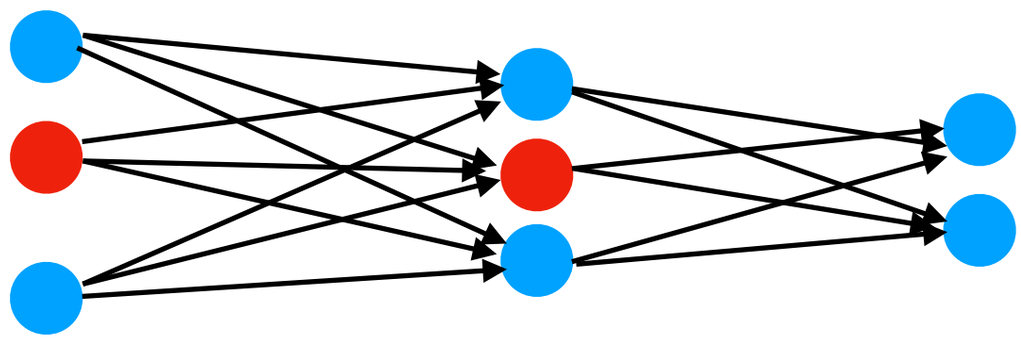
- Suppose we are training the NN shown below:



Dropout

- Suppose we are training the NN shown below:

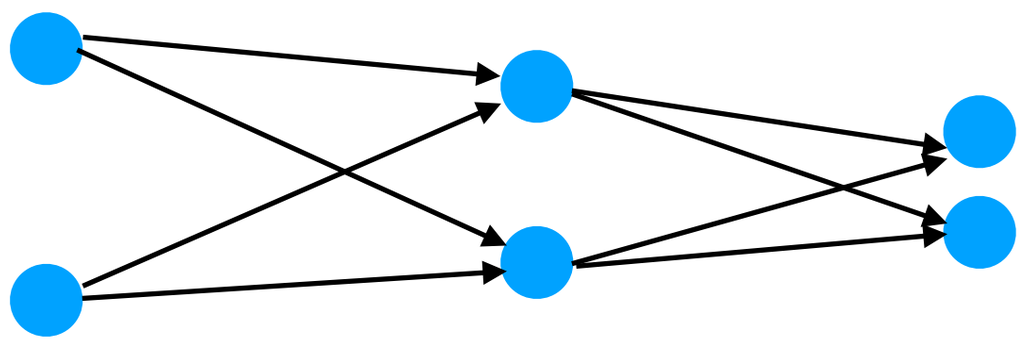
- For each step of SGD, we randomly select (with “keep” probability *p*) some of the input and hidden neurons (*not* the output neurons).



Dropout

- Suppose we are training the NN shown below:

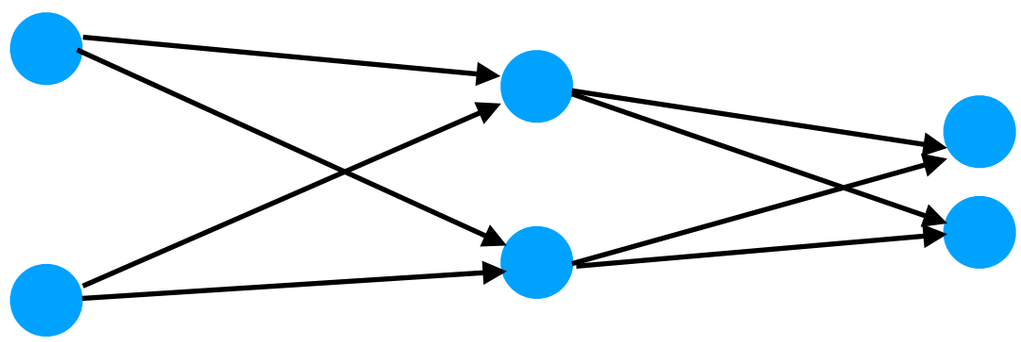
- We then remove these neurons and perform forward-propagation on the reduced network.



Dropout

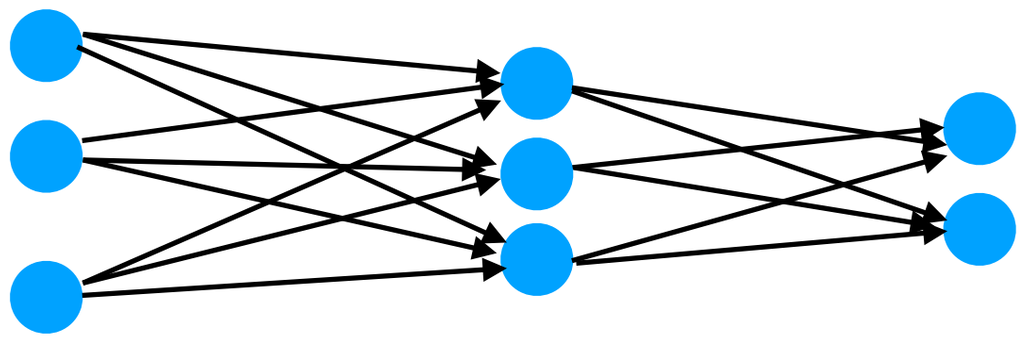
- Suppose we are training the NN shown below:

- During back-propagation, we adjust the weights of *only* those neurons that were retained in the reduced network.



Dropout

- We then replace the neurons we had removed and resume training. (During the next SGD iteration, we will randomly select *another* set of neurons to remove, etc.)

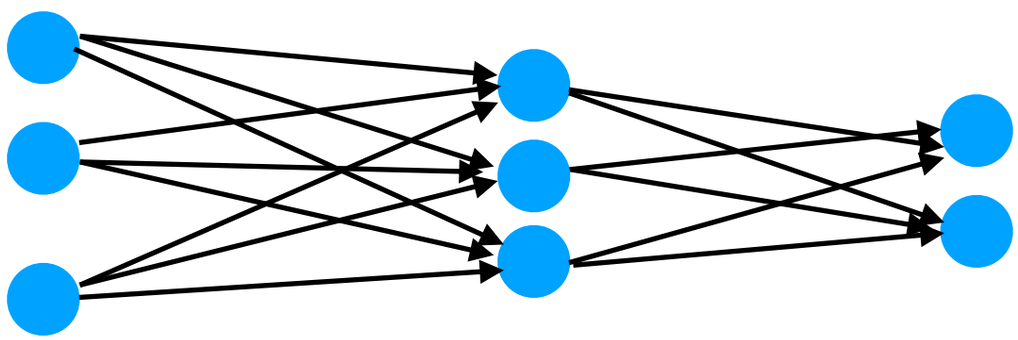


Dropout: example

- Suppose the weights are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

(For simplicity, assume that $\mathbf{b}^{(1)}=\mathbf{b}^{(2)}=0$.)

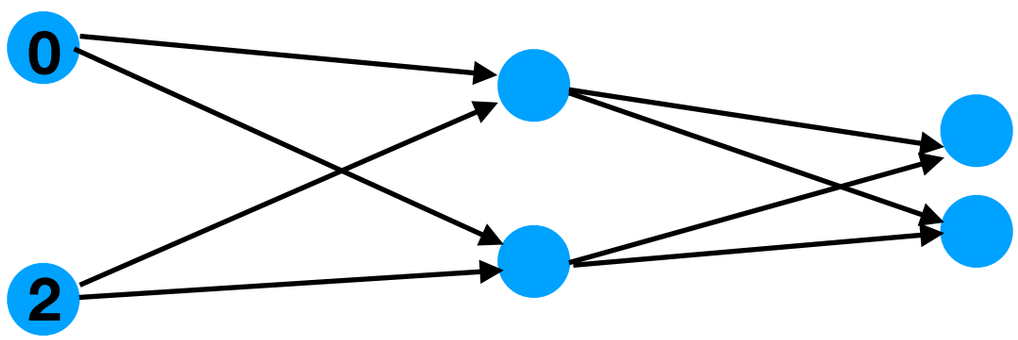


Dropout: example

- Suppose the weights are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- If we drop the red neurons, then we will obtain $\hat{\mathbf{y}}=[60, 132]^T$ for the input $x=[0, 1, 2]^T$ during forward-propagation.

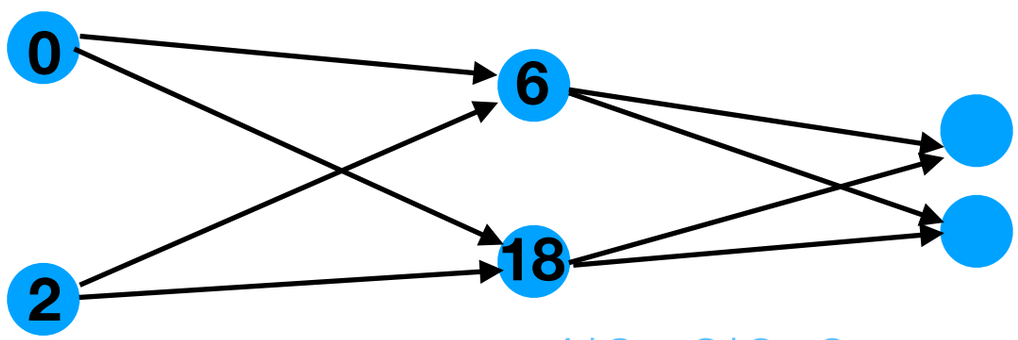


Dropout: example

- Suppose the weights are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- If we drop the red neurons, then we will obtain $\hat{\mathbf{y}}=[60, 132]^T$ for the input $x=[0, 1, 2]^T$ during forward-propagation.

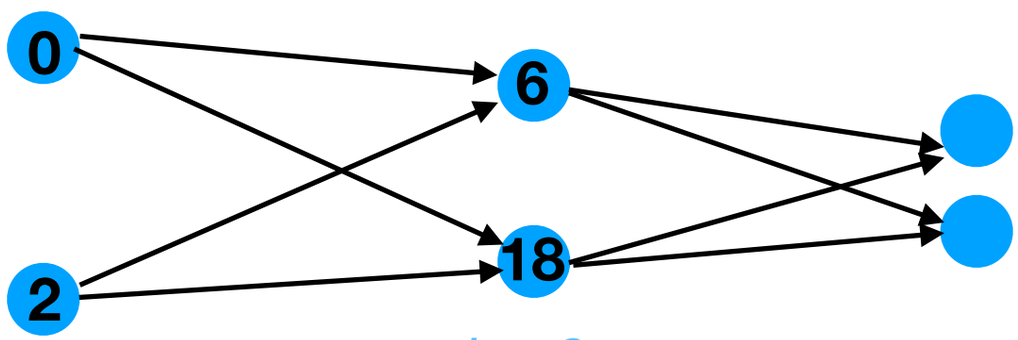


Dropout: example

- Suppose the weights are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- If we drop the red neurons, then we will obtain $\hat{\mathbf{y}}=[60, 132]^T$ for the input $x=[0, 1, 2]^T$ during forward-propagation.

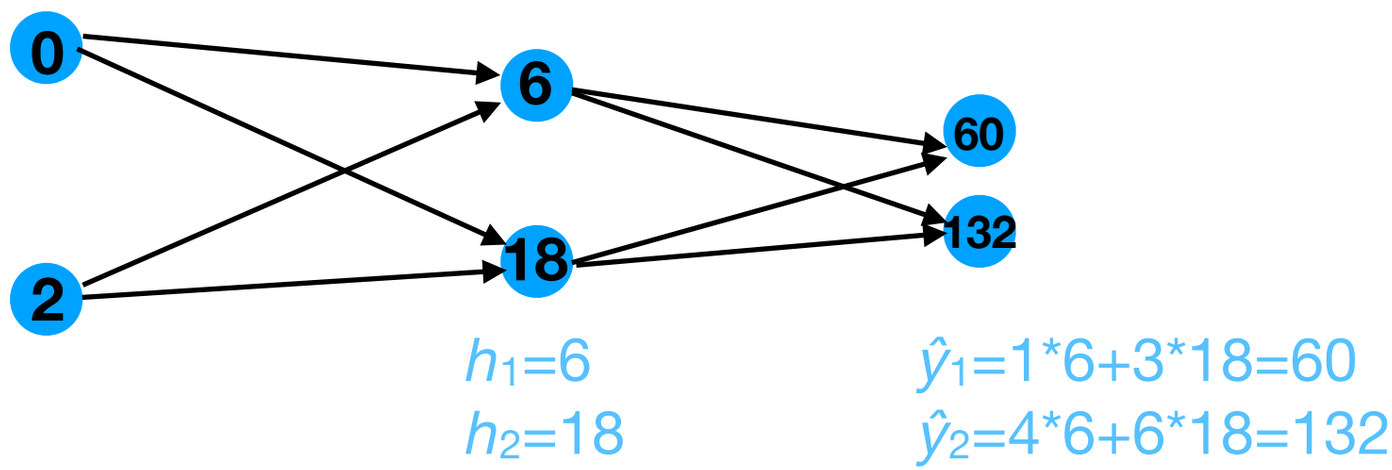


Dropout: example

- Suppose the weights are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- If we drop the red neurons, then we will obtain $\hat{\mathbf{y}}=[60, 132]^T$ for the input $x=[0, 1, 2]^T$ during forward-propagation.

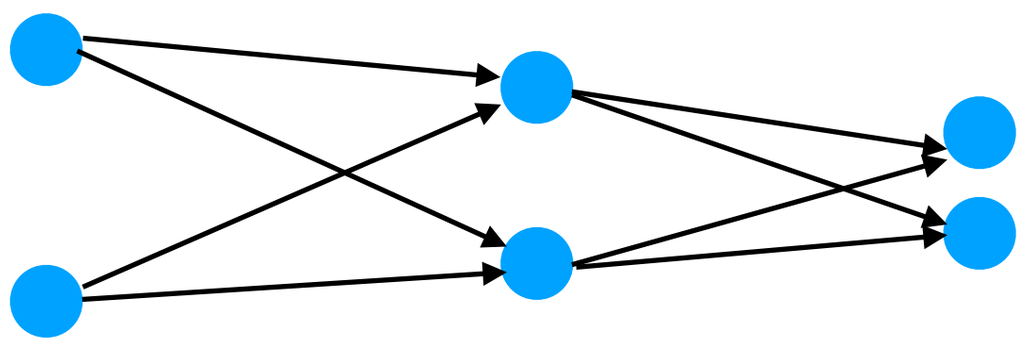


Dropout: example

- Suppose the weights are:

$$\mathbf{w}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \mathbf{w}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- During back-propagation, we will update the weights of only those neurons that were not removed.



Dropout: why helpful?

- There are two main explanations for why dropout helps improve the accuracy of neural networks:

- Symmetry breaking.

- Ensemble of many smaller networks.

Symmetry breaking

- Recall exercise 3 from Lecture 21 about weight initialization:
    - Suppose that each weight matrix & bias vector consists of the same row *repeated many times*.
        - What will happen during SGD?
- The problem was that, when the rows of each weight matrix were initialized to be equal, the gradient updates for each row were also equal.
    - All the rows of each weight matrix moved in “lockstep”.

Symmetry breaking

- Recall exercise 3 from Lecture 21 about weight initialization:
    - Suppose that each weight matrix & bias vector consists of the same row *repeated many times*.
        - What will happen during SGD?
- One of the reasons we initialize weights randomly is to break symmetry between them, so they learn to produce independent values in the subsequent hidden layer.

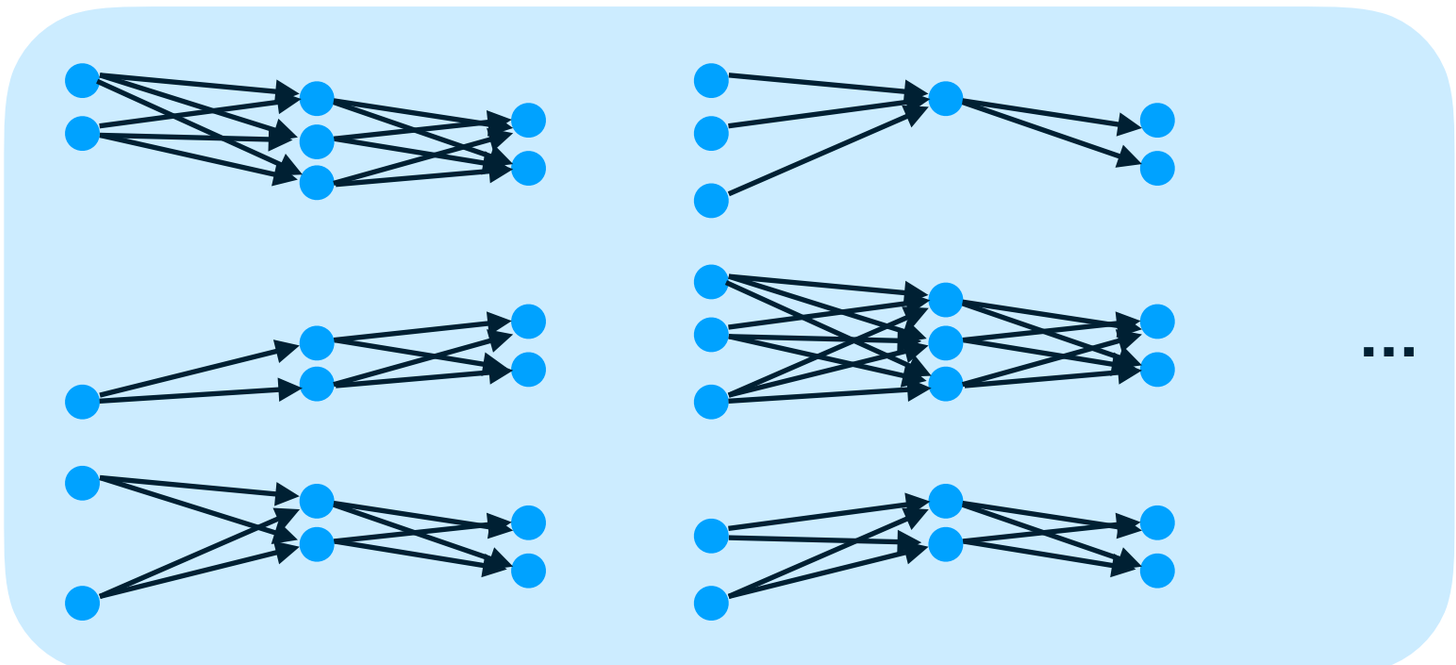
Symmetry breaking

- Recall exercise 3 from Lecture 21 about weight initialization:
    - Suppose that each weight matrix & bias vector consists of the same row *repeated many times*.
        - What will happen during SGD?
- Dropout can also help break symmetry since only some of the elements of each weight matrix are updated during each SGD iteration.

Ensemble of many smaller networks

- Dropout-based NN training can be seen as approximating a large ensemble of many smaller networks.

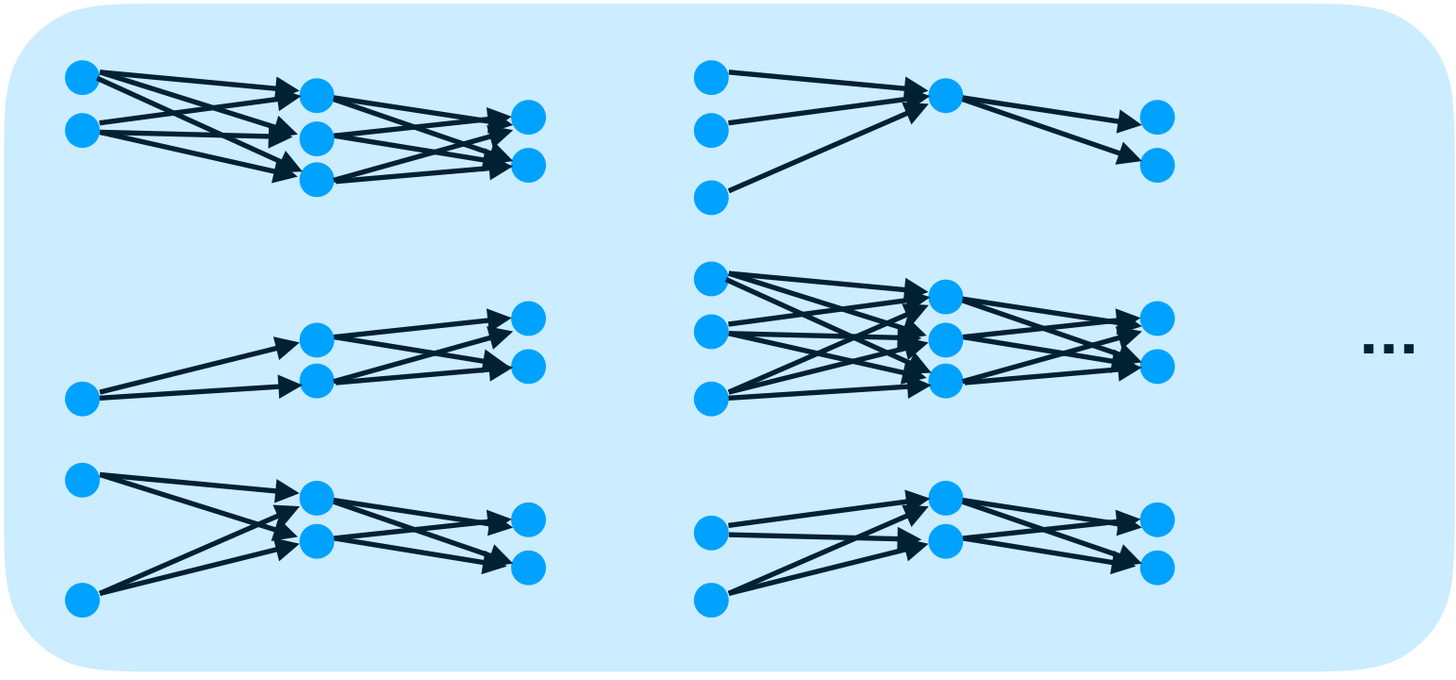
- Each member of the ensemble arises by randomly dropping some of the whole network's neurons:



Ensemble of many smaller networks

- At the end of SGD training, the final network approximates the average prediction over all members of the ensemble.

- Caveat: each member of the ensemble is constrained to *share the same weights* with all other members.



Ensemble of many networks

Data augmentation

Data augmentation

- The more training data you have, the less is the risk of overfitting.
- Unfortunately, training data are often hard to find.
- Can we synthesize new training examples automatically?

Data augmentation

- **Data augmentation** is the creation of new examples based on existing ones.

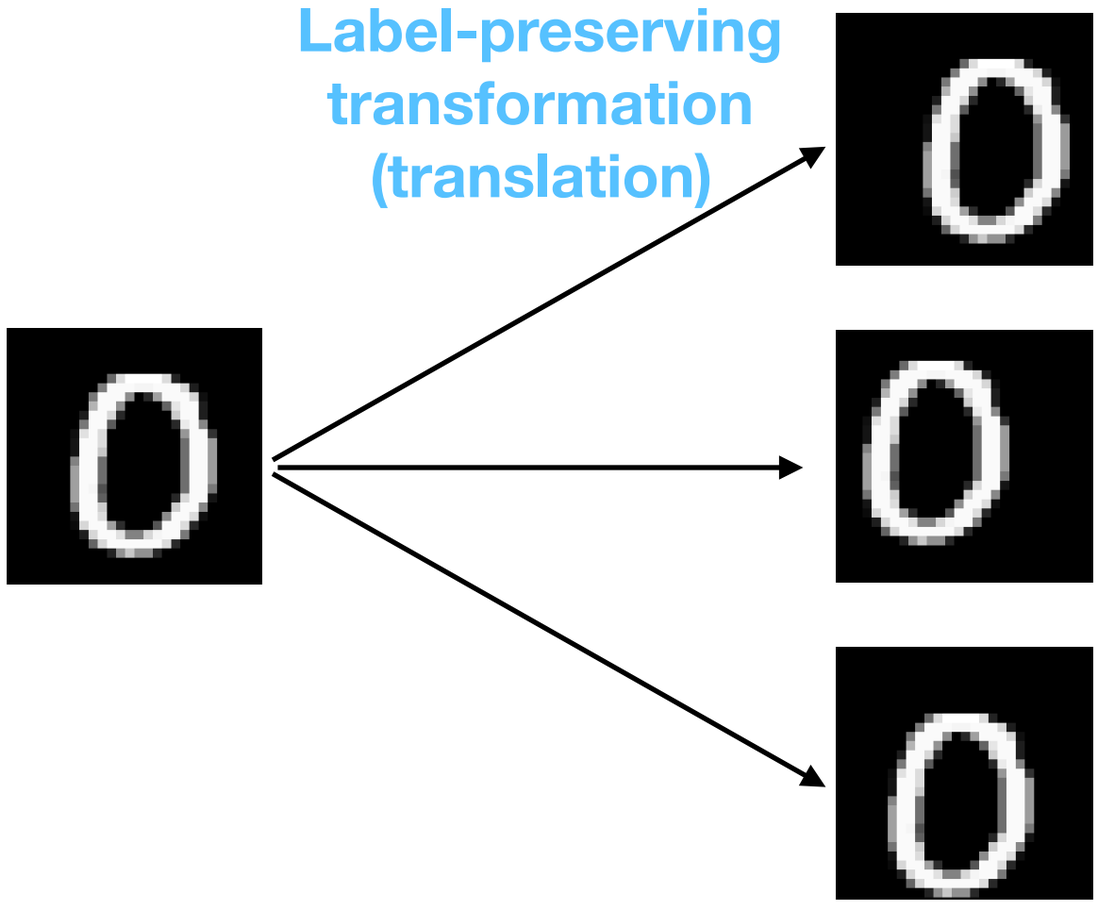
- If we can alter an existing training example without affecting its associated label, then we can generate many new training examples and train on them.

Data augmentation

- Several commonly used methods of data augmentation:
  - Adding noise to existing examples (e.g., Gaussian, Laplacian).
  - Geometric transformations (e.g., flip left/right, rotate, translate).

Example: translation

- From an existing MNIST image, translate all the pixels by some random amount (dx, dy).



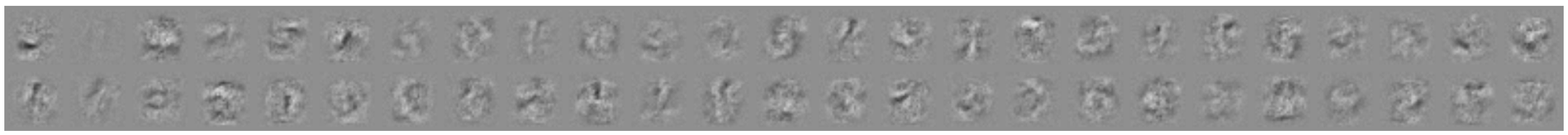
Example: translation

- Data augmentation via translation encourages the NN to learn **translation-invariant** features — they are useful for classification no matter where in the image they occur.

Example: translation

- Here are the weights $\mathbf{W}^{(1)}$ (transformed to 100x28x28) of a MNIST classification network **without** data augmentation:

Acc=98.07



Example: translation

- Here are the weights $\mathbf{W}^{(1)}$ (transformed to 100x28x28) of a MNIST classification network **with** data augmentation:

Acc=98.44

- Compared to the previously shown weights, these show visually more well-defined contours.

Multi-task learning