

ECE521 Lectures 9 & 10

Neural Networks

Deep Learning



UNIVERSITY OF
TORONTO

Outline

- **Multi-class classification**
- Learning multi-layer neural networks
- Bag-of-tricks for deep neural networks
 - Local minimums and initialization
 - Early stopping and regularization
 - Dataset normalization

Measuring distance in probability space

- We learnt that the squared L2 distance is an important concept that captures the natural distance measure between the two points in Euclidean space. The Kullback-Leibler divergence is an equally important concept that is an appropriate distance measure between two probability distributions:

$$KL(Q\|P) = \sum_x Q(x) \log \frac{Q(x)}{P(x)}$$

- KL divergence is an asymmetric distance function: $KL(Q\|P) \neq KL(P\|Q)$
- KL divergence is also nonnegative for any two distributions

Binary cross-entropy loss

- Consider the Q distribution to be the discrete probability distribution of the observed binary labels $t \in \{0, 1\}$ in the training dataset
 - $Q(t = 1|\mathbf{x}) = t$ is either zero or one depending on the training example. (The dataset is observed, so there is no randomness)
- Choose the P distribution to be the model's prediction: $P(t = 1|\mathbf{x}) = \hat{y}(\mathbf{x})$. The KL divergence between Q and P is in fact the cross-entropy loss:

$$\begin{aligned} KL(Q||P) &= \sum_t Q(t|\mathbf{x}) \log \frac{Q(t|\mathbf{x})}{P(t|\mathbf{x})} = - \sum_t Q(t|\mathbf{x}) \log P(t|\mathbf{x}) + \boxed{\sum_t Q(t|\mathbf{x}) \log Q(t|\mathbf{x})} \\ &= - Q(t = 1|\mathbf{x}) \log P(t = 1|\mathbf{x}) - Q(t = 0|\mathbf{x}) \log P(t = 0|\mathbf{x}) \\ &= \underbrace{- t \log \hat{y} - (1 - t) \log(1 - \hat{y})}_{\text{cross entropy}} \end{aligned}$$

← this is zero
 $0 * \log(0) \rightarrow 0$

- The cross-entropy loss function measures the distance between the empirical data distribution and the model predictive distribution

Multi-class cross-entropy loss and softmax

- It is easy to use the KL divergence interpretation to generalize the cross-entropy loss to a multi-class scenario:
 - Let there be K classes, with class labels $t \in \{1, \dots, K\}$
 - The multi-class cross entropy loss can be written using indicator function $I(\cdot)$:

$$KL(Q\|P) = \sum_t Q(t|\mathbf{x}) \log \frac{Q(t|\mathbf{x})}{P(t|\mathbf{x})} = - \sum_{k=1}^K I(k, t_{class}) \log P(t = k|\mathbf{x})$$

- Similarly, the multi-class generalization of the sigmoid function is the *softmax function*. The multi-class predictive distribution becomes:

$$P(t = k|\mathbf{x}) = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

Here z_k are the outputs of a neural network or linear regression

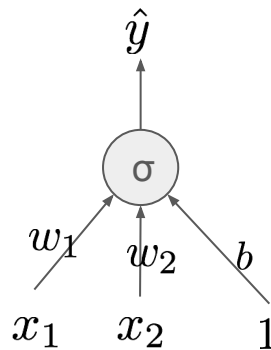
Outline

- Multi-class classification
- **Learning multi-layer neural networks**
- Bag-of-tricks for deep neural networks
 - Local minimums and initialization
 - Early stopping and regularization
 - Dataset normalization

Example 1: representing digital circuits with neural networks

- Let us look at a simple example of a soft OR gate simulated by a neural network:
 - Use a single sigmoid neuron with two inputs and a bias unit.

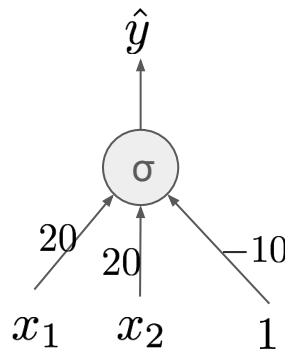
	$x_2=0$	$x_2=1$
$x_1=0$	0	1
$x_1=1$	1	1



Example 1: representing digital circuits with neural networks

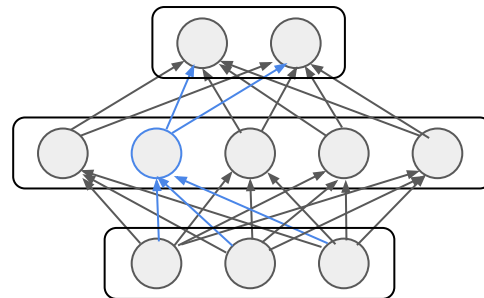
- Let us look at a simple example of a soft OR gate simulated by a neural network:
 - Use a single sigmoid neuron with two inputs and a bias unit.
 - One possible solution is to use the bias as a threshold while setting w_1 and w_2 to be large positive values. When either of the inputs is non-zero, the sigmoid neuron will be turned on and the output will be 1.

	$x_2=0$	$x_2=1$
$x_1=0$	0	1
$x_1=1$	1	1



Learning fully connected multi-layer neural networks

- There are many choices when “crafting” the architecture of a neural network. The fully connected multi-layer NN is the most general multi-layer NN:
 - Each neuron has its incoming weights connected to *all* the neurons from the previous layer and its outgoing weights connected to *all* the neurons in the next layer.
- Fully connected network is the go-to architecture choice if we do not have any additional information about the dataset.
 - After choosing the network architecture, there are a few more engineering choices: #hidden units, #layers, the type of activation function.
 - The output units type: linear, logistic or softmax are determined by output tasks, i.e. regression or classification task



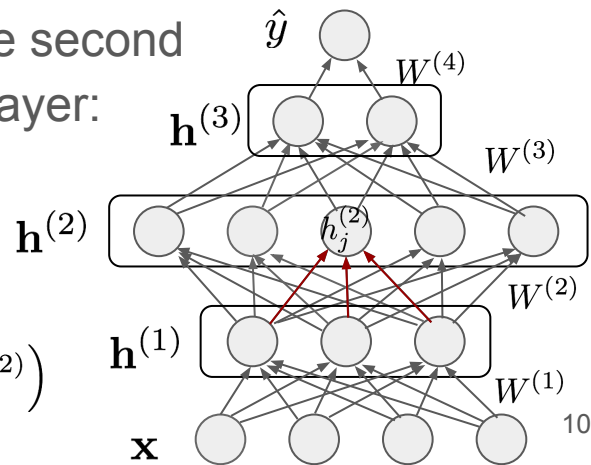
Learning fully connected multi-layer neural networks

- Consider a fully connected neural network with 3 hidden layers:
 - The input to the neural network is an N -dimensional vector \mathbf{x} . There are H_1 , H_2 , and H_3 hidden units in the three hidden layers. We use superscript to index the layers.
 - There are four weight matrices among the hidden layers, e.g. $W^{(2)} \in \mathbb{R}^{H_2 \times H_1}$, $b^{(2)} \in \mathbb{R}^{H_2}$
 - The j th row of the weight matrix $W^{(2)}$ is denoted as $\mathbf{W}_j^{(2)} \in \mathbb{R}^{H_1}$
- The hidden activation of the j th hidden unit $h_j^{(2)}$ in the second hidden layer is the weighted sum of the first hidden layer:

$$h_j^{(2)} = \phi(z_j^{(2)}) = \phi\left(\sum_i w_{ij}^{(2)} h_i^{(1)} + b_j^{(2)}\right) = \phi\left(W_j^{(2)T} \mathbf{h}^{(1)} + b_j^{(2)}\right)$$

- We can use vector notation to express the hidden vector:

$$\mathbf{h}^{(2)} = \begin{bmatrix} h_1^{(2)} \\ \vdots \\ h_{H_2}^{(2)} \end{bmatrix} = \begin{bmatrix} \phi(z_1^{(2)}) \\ \vdots \\ \phi(z_{H_2}^{(2)}) \end{bmatrix} = \phi\left(\begin{bmatrix} W_1^{(2)T} \\ \vdots \\ W_{H_2}^{(2)T} \end{bmatrix} \mathbf{h}^{(1)} + b^{(2)}\right) = \phi\left(W^{(2)} \mathbf{h}^{(1)} + b^{(2)}\right)$$



Learning fully connected multi-layer neural networks

- For a single data point, we can write the the hidden activations of the fully connected neural network as a recursive computation using the vector notation:

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + b^{(1)}, \quad \mathbf{h}^{(1)} = \phi\left(\mathbf{z}^{(1)}\right)$$

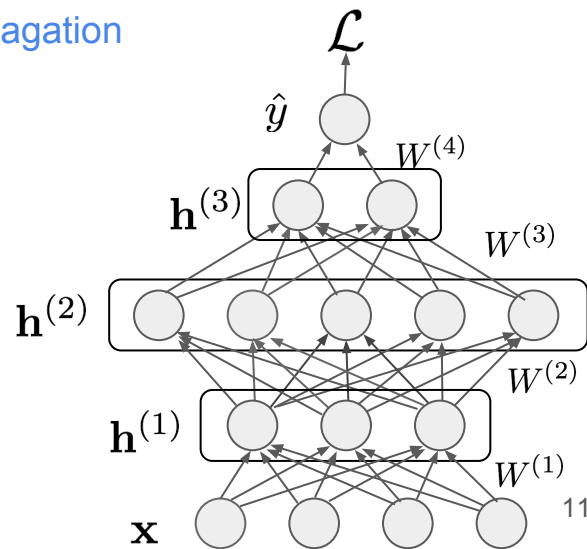
$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{h}^{(1)} + b^{(2)}, \quad \mathbf{h}^{(2)} = \phi\left(\mathbf{z}^{(2)}\right)$$

$$\mathbf{z}^{(3)} = W^{(3)}\mathbf{h}^{(2)} + b^{(3)}, \quad \mathbf{h}^{(3)} = \phi\left(\mathbf{z}^{(3)}\right)$$

$$\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}, \quad \hat{y} = f\left(\mathbf{z}^{(4)}\right)$$

- $f()$ is the output activation function
- The output of the network is then used to compute the loss function on the training data

forward
propagation



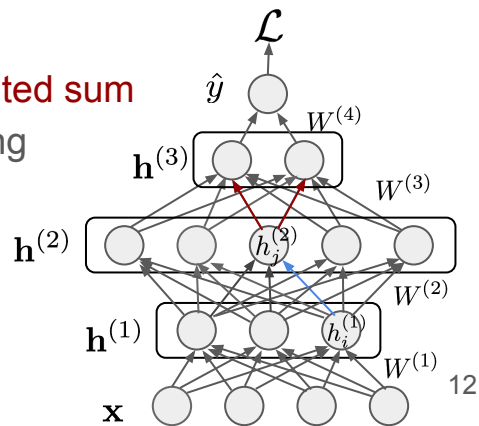
Learning fully connected multi-layer neural networks

- Learning neural networks using stochastic gradient descent requires the gradient of the weight matrices from each hidden layer.
 - Let us consider the gradient of the loss for a single training example. The gradient w.r.t. the incoming weights $w_{ij}^{(2)}$ of the j th hidden unit in the second layer is the *product* of the hidden activation from layer 1 *and* the partial derivative w.r.t. z_j . Remember: $h_j^{(2)} = \phi(z_j^{(2)}) = \phi\left(\sum_i w_{ij}^{(2)} h_i^{(1)} + b_j^{(2)}\right)$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial w_{ij}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(2)}} h_i^{(1)}$$

- The partial derivative w.r.t. z_j in the second hidden layer is the **weighted sum of the partial derivatives** from the third layer, weighted by the outgoing weights of the j th hidden units:

$$\frac{\partial \mathcal{L}}{\partial z_j^{(2)}} = \frac{\partial \mathcal{L}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial h_j^{(2)}} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} w_{ji}^{(3)} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}$$



Learning fully connected multi-layer neural networks

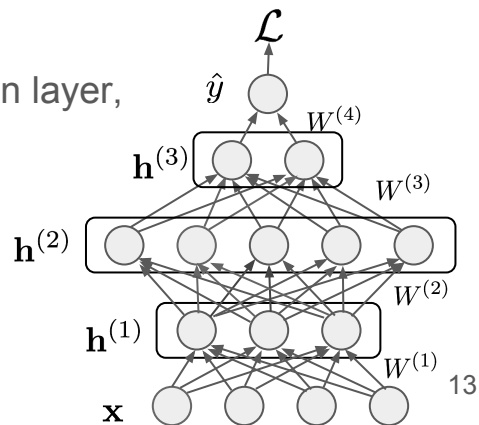
- Similar to the hidden-activation computation (slide 10), the weighted sum of the partial derivatives can be rewritten using vector notation:

$$\frac{\partial \mathcal{L}}{\partial z_j^{(2)}} = \left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} w_{ji}^{(3)} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \left(\mathcal{W}_{j \cdot}^{(3)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}$$

$$\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} = \begin{bmatrix} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}} & 0 & \dots & 0 \\ \vdots & \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} & \vdots & \vdots \\ 0 & \dots & \frac{\partial h_{H_2}^{(2)}}{\partial z_{H_2}^{(2)}} \end{bmatrix} = \text{diag} \left\{ \begin{bmatrix} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}} \\ \vdots \\ \frac{\partial h_{H_2}^{(2)}}{\partial z_{H_2}^{(2)}} \end{bmatrix} \right\}$$

- Here, $\mathcal{W}_{j \cdot}^{(3)}$ is the j th column of the weight matrix $W^{(3)}$
- To express the partial derivatives w.r.t. \mathbf{z} for the entire second hidden layer, we can use a matrix-vector product:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial z_{H_2}^{(2)}} \end{bmatrix} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(\begin{bmatrix} \mathcal{W}_{1 \cdot}^{(3)T} \\ \vdots \\ \mathcal{W}_{H_2 \cdot}^{(3)T} \end{bmatrix} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right) = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(W^{(3)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right)$$



Learning fully connected multi-layer neural networks

- For a single training datum, computing the gradient w.r.t. the weight matrices is also a recursive procedure:

- Remember: $\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}$, $\hat{y} = f(\mathbf{z}^{(4)})$
- Back-propagation is similar to running the neural network backwards using the transpose of the weight matrices

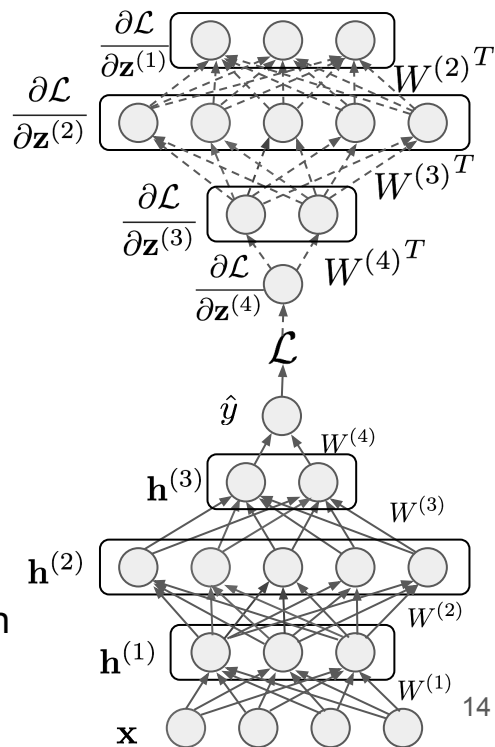
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} = \frac{\partial \hat{y}}{\partial \mathbf{z}^{(4)}} \frac{\partial \mathcal{L}}{\partial \hat{y}}, \quad \frac{\partial \mathcal{L}}{\partial W^{(4)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} \mathbf{h}^{(3)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} = \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{z}^{(3)}} \left(W^{(4)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(3)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \mathbf{h}^{(2)T} \quad \text{back-propagation}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(W^{(3)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} \mathbf{h}^{(1)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}} \left(W^{(2)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \mathbf{x}^T$$

What about the expression for the bias units?

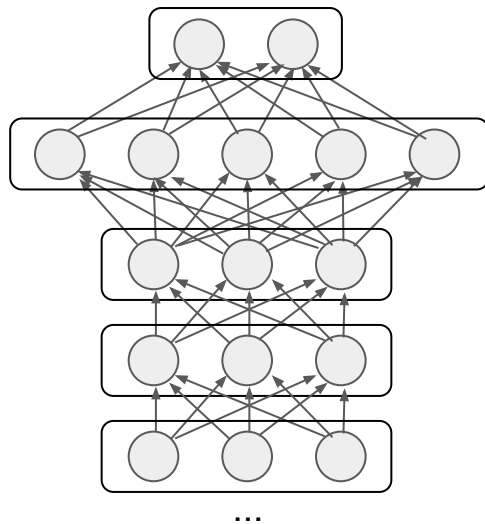


Outline

- Multi-class classification
- Learning multi-layer neural networks
- **Bag-of-tricks for deep neural networks**
 - Local minimums and initialization
 - Early stopping and regularization
 - Dataset normalization

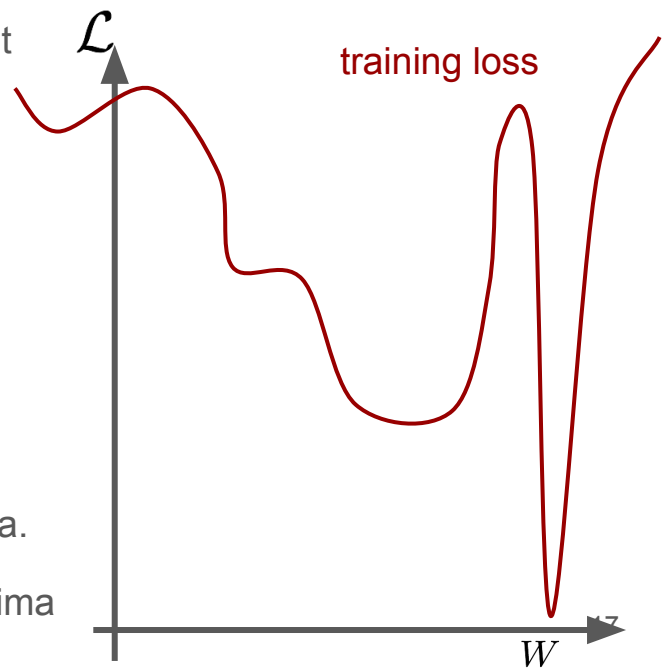
Hyper-parameters

- At the beginning, there were the choices:
 - How many hidden units to use in each hidden layer?
 - How many layers in total?
 - Which hidden activation function?
- Good answer: decide these hyper-parameters using validation set.
- Best practical answer:
 - Around 500-2000 hidden units
 - 2-3 layers
 - Choosing ReLU often leads to fast convergence



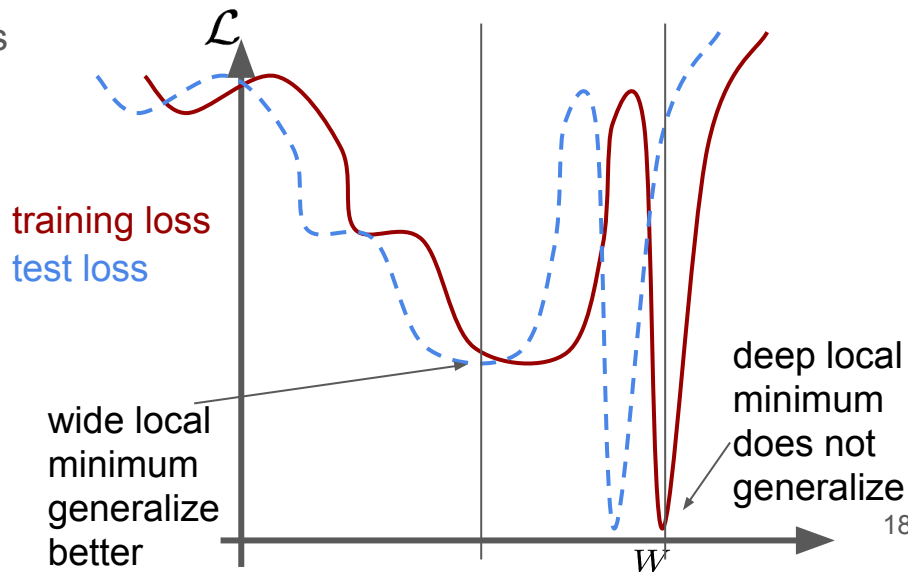
Parameter initialization

- The loss functions for neural networks w.r.t. the weight matrices in general are non-convex.
 - Different weight-initialization schemes can lead to significant differences in final performance.
 - Use random initialization (e.g. Gaussian with std. 0.01) and avoid constant initialization
- Non-convex optimization is not crazy. Most of the local minima in a neural network's loss function are not bad.
 - As long as the model has enough capacity to model the data.
 - Stochastic gradient descent can usually find good local minima



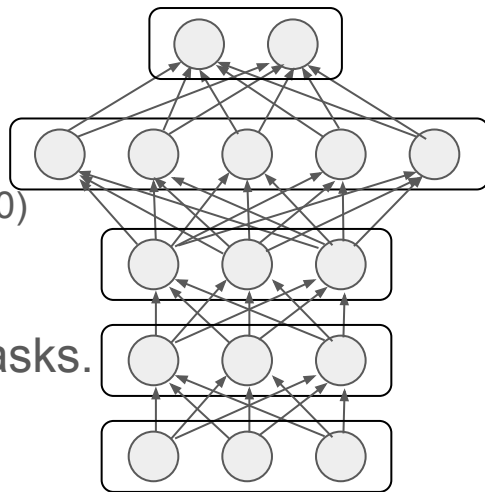
Some local optima generalize better than others

- Consider two neural networks that each achieve a low error rate on the training set. We prefer the model that learns the underlying statistical patterns of the data and can generalize to unseen examples during test time.
 - The test loss function and the training loss are almost always slightly different.
 - Wide shallow basins can typically generalize better than deep narrow local minima.
 - The subsampling noise from SGD helps to find shallow basins.



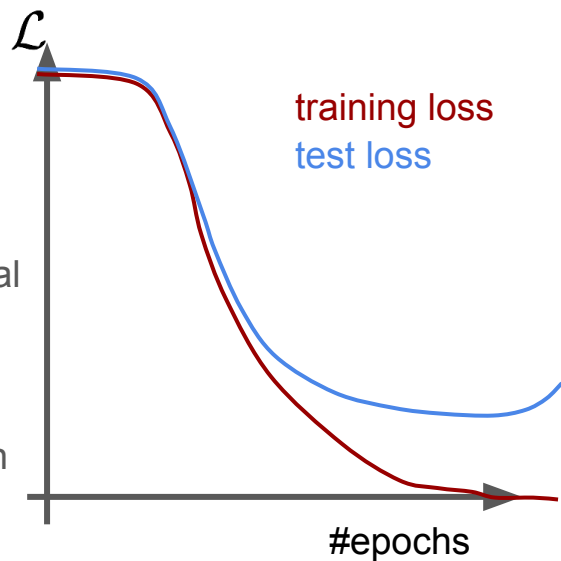
Careful initialization

- For really deep neural networks (> 5 layers), random initialization from a constant-variance Gaussian noise will not work well. The back-propagated partial derivatives will likely be too small to learn anything useful.
 - Simple fix: initialize the weight matrices to identity matrices if you can. (Le, Jaitly and Hinton, 2015)
 - More elaborate fix: adapt the std. of the zero-mean Gaussian initialization to be $1/\sqrt{\#\text{inputs} \times \#\text{outputs}}$ aka *Xavier initialization* (Xavier Glorot and Yoshua Bengio, 2010)
- Oftentimes it is beneficial to initialize the weights from another model that was “pre-trained” on some other tasks.
 - Initialize the model from an auto-encoder or ImageNet models.
 - Weight matrices in the early layers are transferable and help generalization. ...



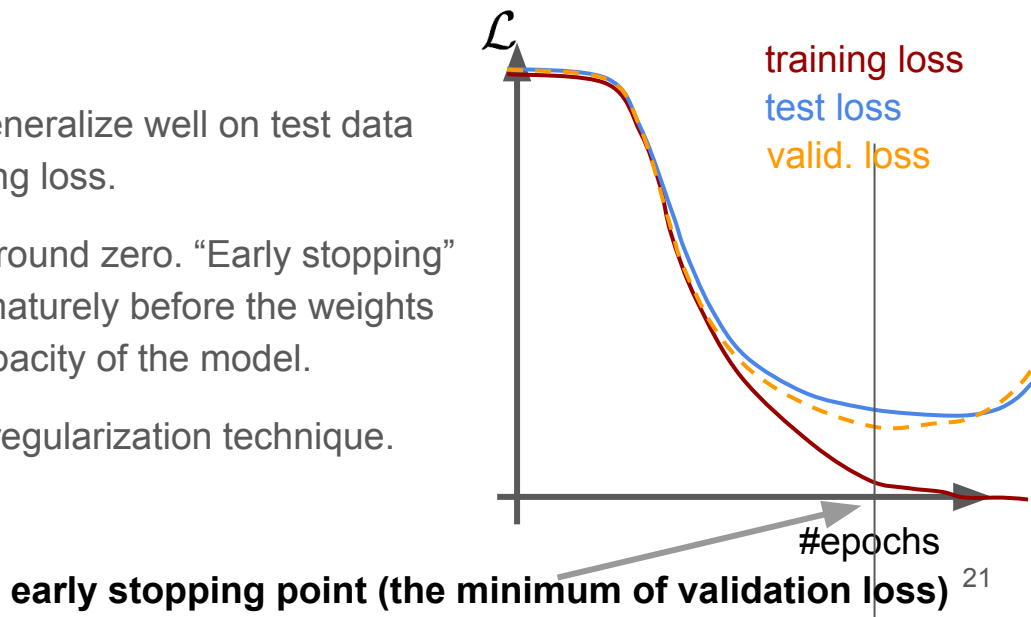
Regularize the capacity of deep neural networks

- Given enough hidden units, neural networks can overfit to the training dataset. Regularizing a neural network is equivalent to restricting its capacity.
 - Reducing the number of hidden units to limit the model capacity has a sound statistical justification: you need more training examples than the number of weights to have enough *statistical power* when estimating the unknown model parameter.
 - Alternatively, one can have an over-parameterized model and deal with overfitting through a very strong regularizer such that most of the weights are close to zero, e.g. using weight decay.
 - In deep learning applications, strong-regularizer approaches often work much better. A good heuristic is to prefer a globally simple prediction function with some irregular local behaviour.



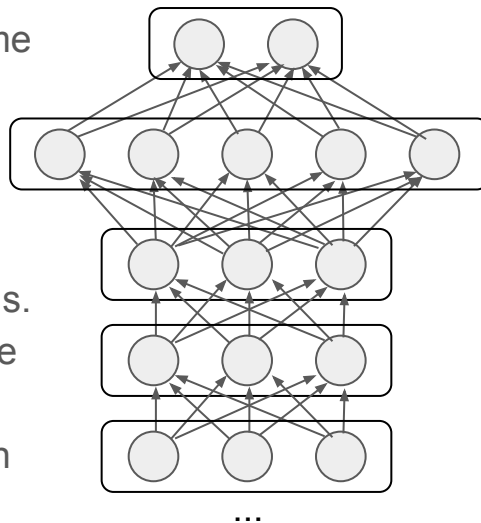
Early stopping

- A very simple trick to ensure that most of the weights are as close to zero as possible involves monitoring the *validation loss*. You stop learning at the minimum of the validation loss curve (aka “early stopping”) before attaining the minimum training loss.
 - The goal of machine learning is to generalize well on test data instead of finding the minimum training loss.
 - The weights are typically initialized around zero. “Early stopping” terminates the learning process prematurely before the weights grow too large and thus limits the capacity of the model.
 - It is by far the most commonly used regularization technique.



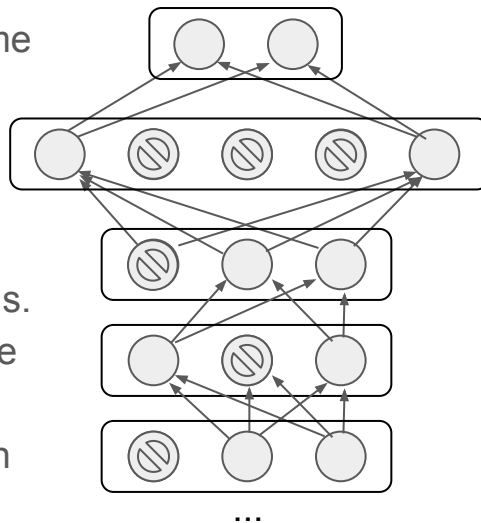
Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
 - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
 - Efficiently shares the statistical power among an ensemble of neural networks.
 - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



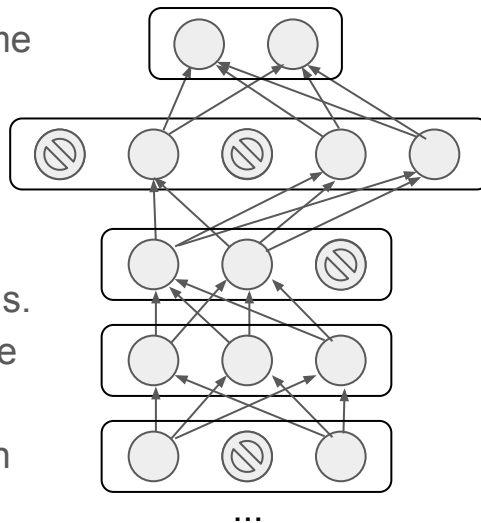
Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
 - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
 - Efficiently shares the statistical power among an ensemble of neural networks.
 - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
 - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
 - Efficiently shares the statistical power among an ensemble of neural networks.
 - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



Summary of regularization options

- Low number of units
- Weight decay
- Early stopping
- Dropout

Dataset normalization

- Learning can often be made easier by pre-processing the training dataset before performing any training.
 - The simplest normalization scheme is to centre the input \mathbf{x} and remove its variance for each input dimension. It fixes the scaling discrepancy among the input dimensions (e.g. removes the units of measurement).
 - For each of the N input dimensions, estimate its mean and variance from the training data and transform the m th training example by:

$$\bar{x}_n^{(m)} = (x_n^{(m)} - \mu_n) / \sigma_n$$

$$\mu_n = \frac{1}{M} \sum_{m=1}^M x_n^{(m)} \quad \sigma_n = \sqrt{\frac{1}{M-1} \sum_{m=1}^M (x_n^{(m)} - \mu_n)^2}$$

- We can further remove the covariance among the input dimensions, i.e. whitening the data using Principal Component Analysis (PCA): next lecture