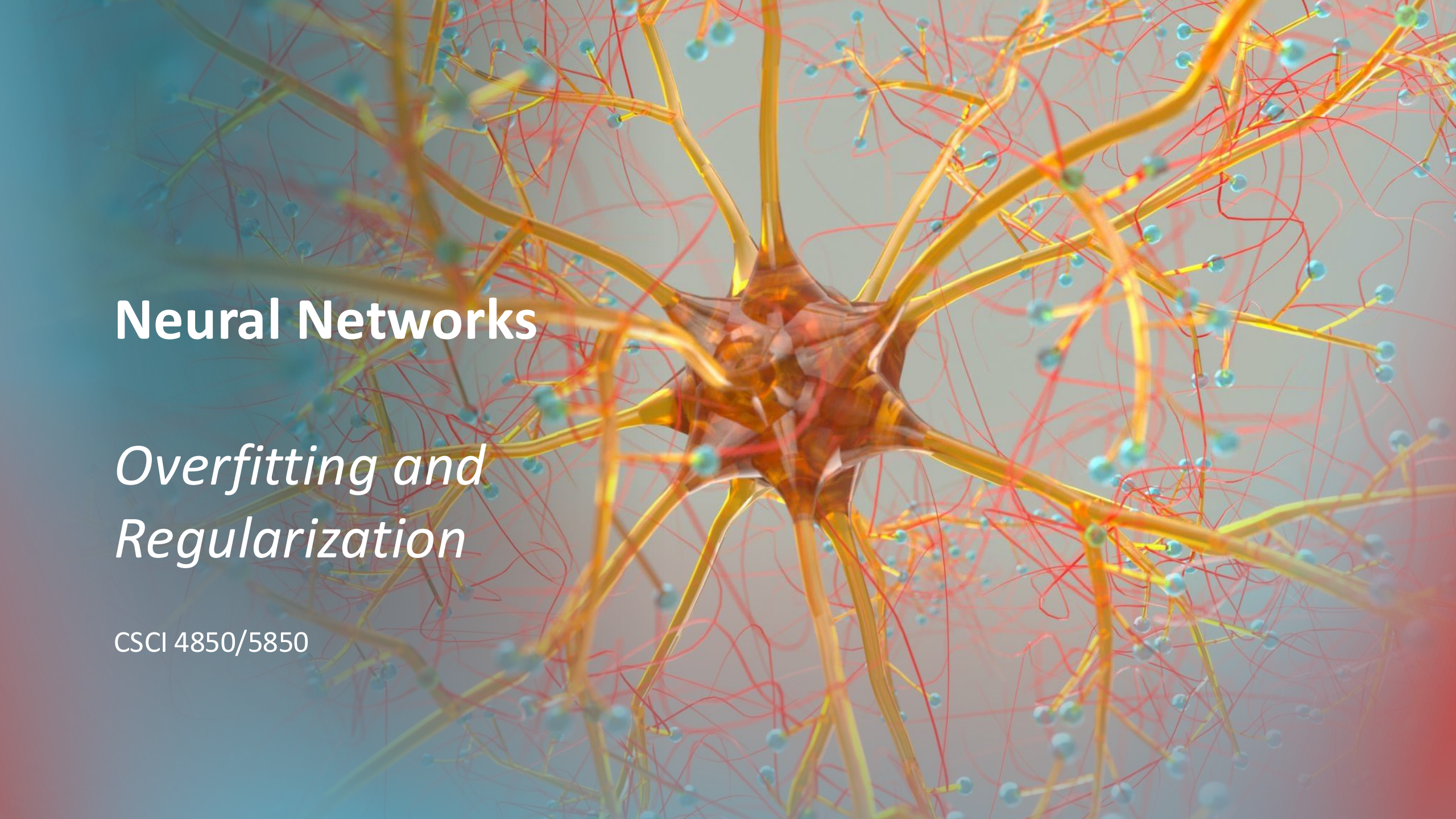


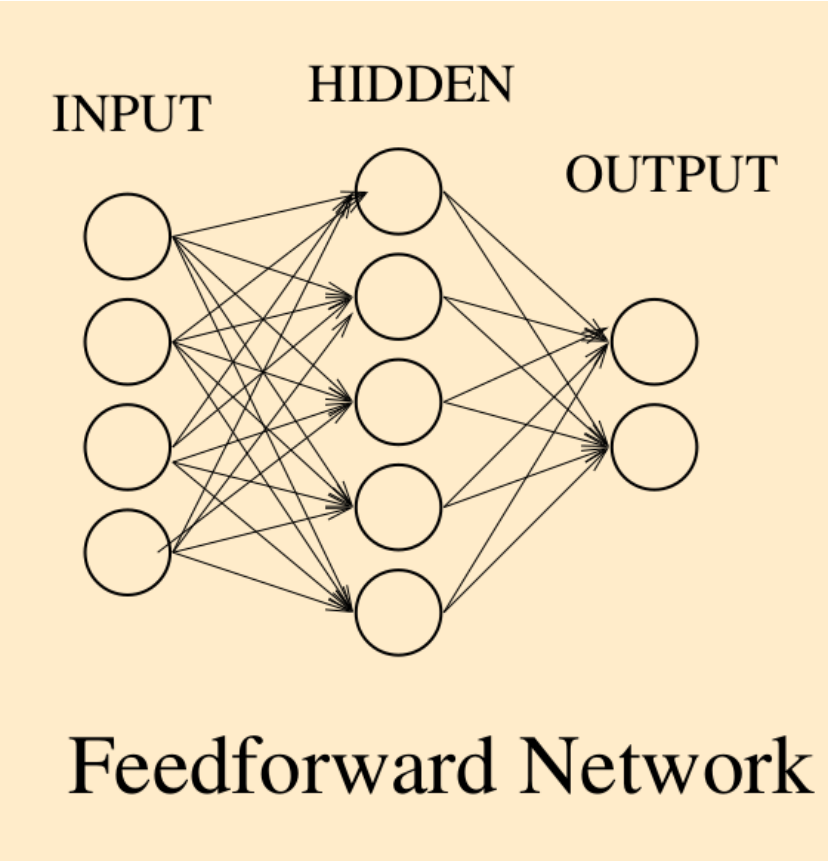
Neural Networks

Overfitting and Regularization

CSCI 4850/5850

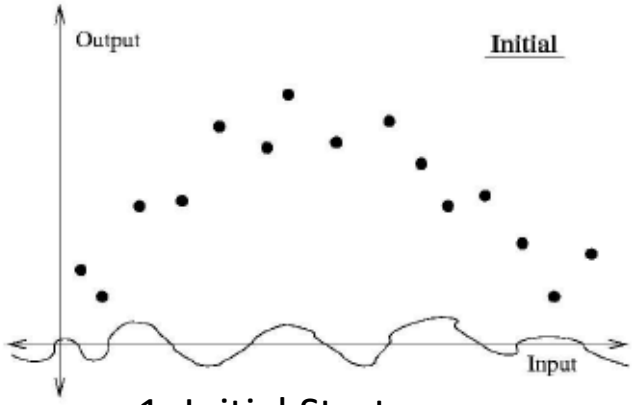
Universal Approximation

- It has been proven that given *enough* units and using only a single layer of hidden units, essentially *any* function can be approximated to an arbitrary degree of precision
- Hornik, Stinchcombe, and White, 1989
- Can **every** kind of function be *learned* in this way?
  - We aim for good generalization, not necessarily precision…
  - The loss function (which we are optimizing) pushes us toward better precision, but not necessarily good generalization (which we are **not** optimizing)
  - How do we help make generalization happen?
  - Can tune the **inductive bias** of the network by adapting *network architecture or data transformation* based on domain knowledge
  - But what else can we do?

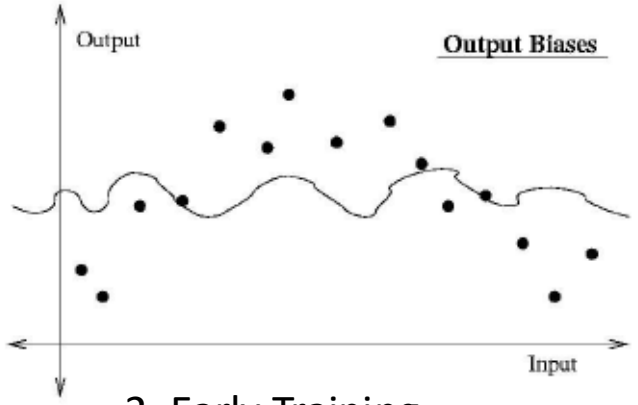


Review: Overfitting and Generalization

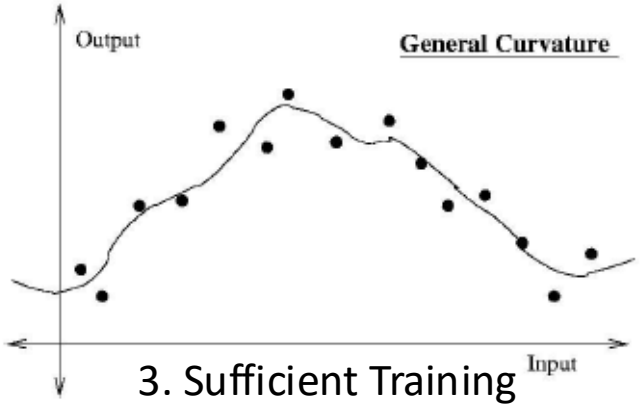
- Biases are learned *early* in the learning process (represent “general” responses)
- Overfitting happens *late* in the training process (represent “specific” responses)



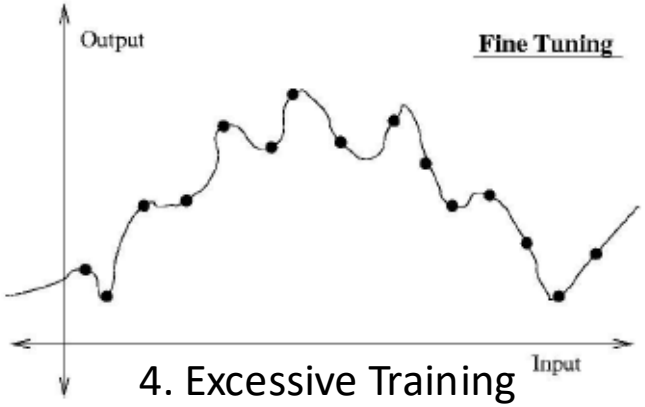
1. Initial Start



2. Early Training



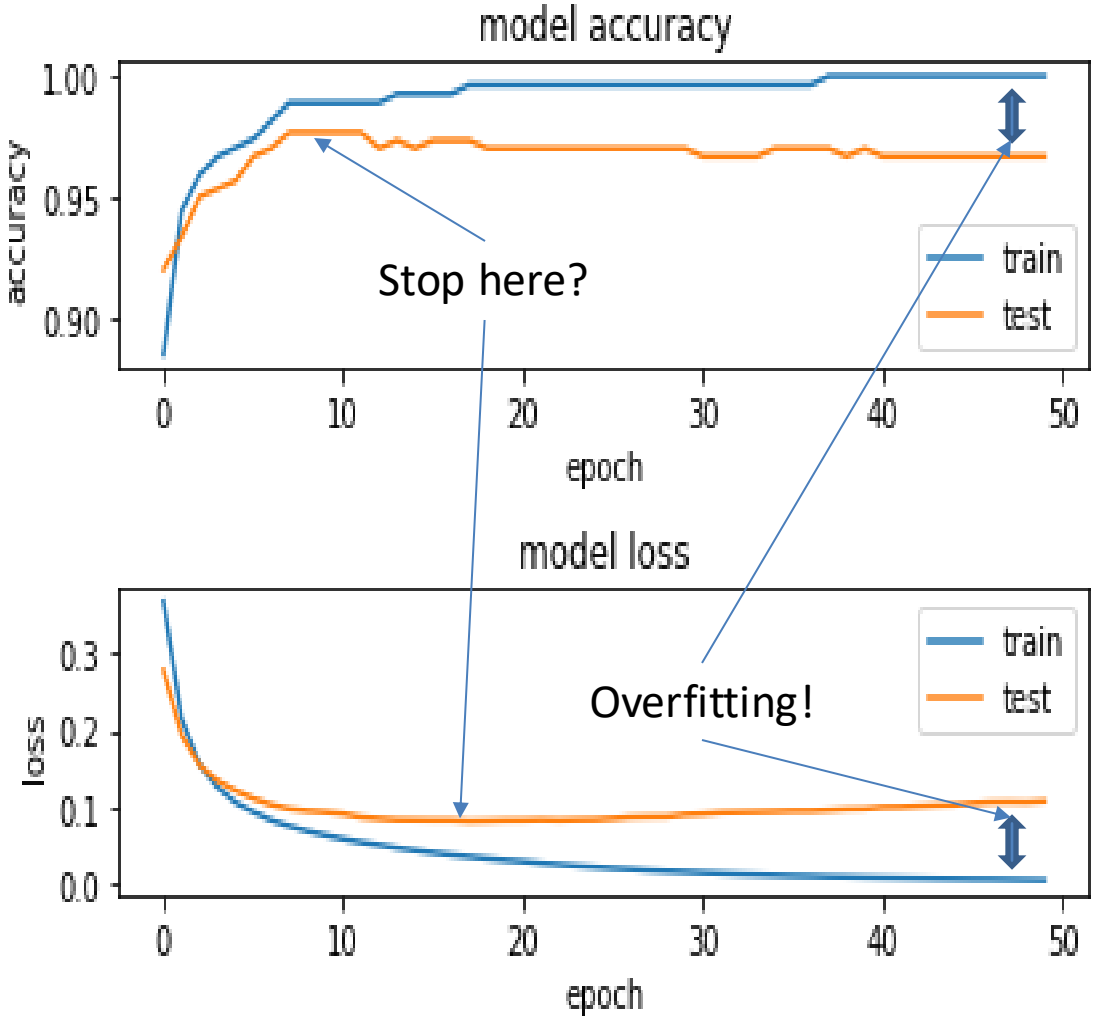
3. Sufficient Training



4. Excessive Training

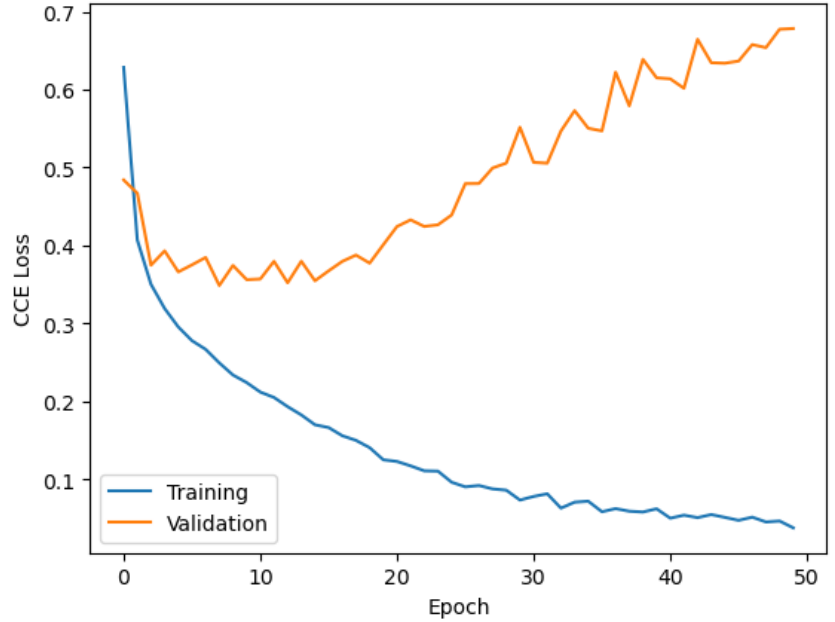
Observing Overfitting in Practice

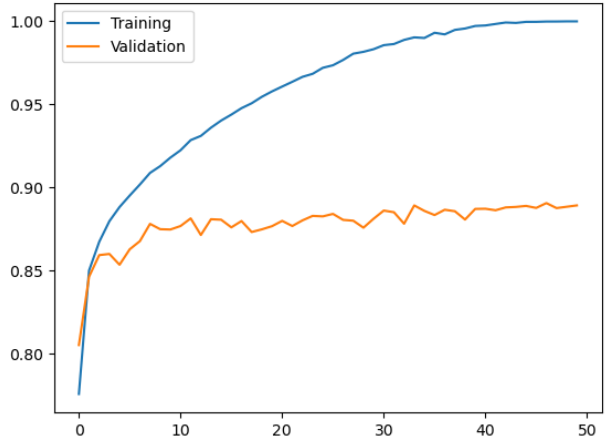
- Must use the **validation set** to check for **overfitting**
- Requires *early stopping* criteria
- *Different* metrics may suggest *different* stopping points…



Second Example: Fashion MNIST

- Deep learning models generally have **greater learning capacity** compared to wide networks.
- Fashion MNIST quickly shows saturation to **100% on the training accuracy** – but stalls at around **88% for the validation accuracy.**
- Continued training will *eventually* force the validation accuracy to **decrease** just like the previous example, but the **validation loss** signaled the problem **earlier in training**.

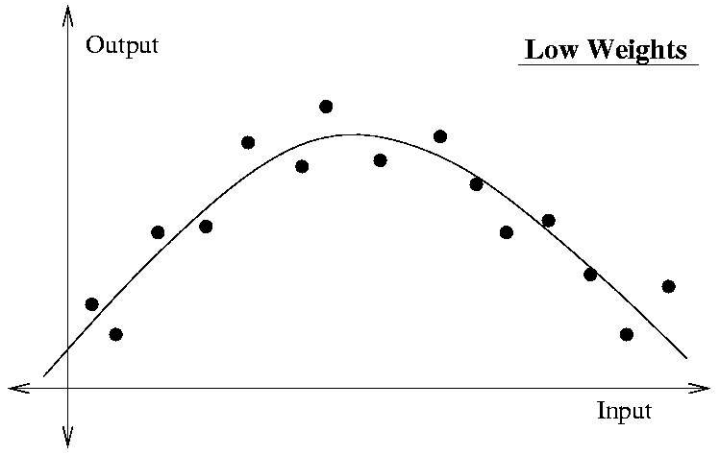




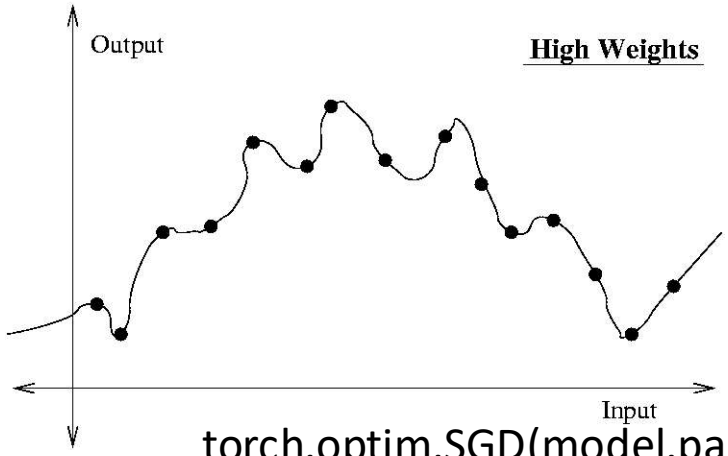
Stopping Overfitting: Regularization

- Add a *preference* term to the **error/loss** function that corrects for the problem
  - We know that our loss function isn't interested in generalization, but we might make it so!
    - $\tilde{E} = E + \nu\Omega$
- Coefficient, ν, determines how much we want to focus on this *generalization preference*
- General approach: compute a new gradient with respect to our **regularized** loss function

Weight Decay – L2 Regularization



Intuition: *overfit* models have *higher magnitude* weights (pushed too far by loss gradient)



- Let Ω be a penalty for high weights…

$$\Omega = \frac{1}{2}\sum_{i,j} w_{ij}^2$$

- Derivative with respect to the new error function:

$$\Delta w_{ij} \propto -\delta_i a_j - \nu w_{ij}$$

- Keeps weights small, *smoother* function learned

torch.optim.SGD(model.parameters(), lr=1e-4, weight_decay=1e-5)

torch.optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-5)

Weight “Elimination” – L1 Regularization

- According to L2 distance (Euclidean), the shortest distance between two points is always a straight line
- According to L1 distance (Manhattan), the shortest distance between two points may be different paths (two blocks north, one block east, one block north = three blocks north, one block east)

$$\Omega = \sum_{i,j} |w_{ij}|$$

- Adding an L1 penalty will make some weights low, but allow others to stay large (sparse connections)

This method is less common and therefore requires customization of the training_step():

```
# define in constructor
self.l1_lambda = 0.01
# use in training_step()
l1_reg = torch.tensor(0., requires_grad=True)
for name, param in self.named_parameters():
    if 'weight' in name:
        l1_reg = l1_reg + torch.norm(param, 1)
loss = loss + self.l1_lambda * l1_reg
```

Activation Regularization

- Rather than impose the penalties on weights, we can impose the penalty to unit *activations*
  - Smaller activations prevent large weight updates
  - Sparse activity patterns prevent overly complicated encodings
- Both L2 or L1 activity regularization could be used…

$$\Omega = \sum_j |a_j|$$

Typically, this form of regularization is not needed when proper normalization methods - BatchNorm() or LayerNorm() - are included in the architecture.

Noisy Inputs/Activations

Another way to get at “averaged” or “smoothed” results is to add noise directly into the network and/or training data

Typically, this is just Gaussian noise layered on the unit activations (not active at test time for consistency of testing)

Regularization: Dropout

Similar idea to L1 activation regularization, but with an element of **chance**

- Set a probability hyperparameter (p) to some fraction for the layer

Each forward pass, some fraction, p, of unit *activations* are randomly set to *zero* (units may randomly turn off!)

Why? Units can no longer "specialize"

- Detecting certain features can't be unit-specific anymore
- Units need to share more information amongst each other since dedicated detectors can't be formed

Dropout generates noisy outputs!

- Different network responses for same input (due to random chance of which units stay on)
- During **training**, activations are *scaled by the dropout fraction* to represent the *average activation* for the given input pattern
- Dropout is **turned off** during **testing** (allows for consistent results)
- Average output represents uncertainty in the function mapping: ad-hoc regularization, but commonly used in deep networks…

Dropout

- Torch implements dropout using a separate layer type which performs the operation of turning off certain activations in the prior layer
  - Think if this as a “filtering” layer where the filter randomly changes on each pattern presentation…

```
# Dropout – in constructor
self.dropout_op = torch.nn.Dropout(p=0.33) # 33% of previous activations -> zero!

# in forward()
y= dropout_op(y)
```

Data Augmentation

- Sentences: present them both forward and backward
- Images: present them with flipped vertical/horizontal orientation, translated, scaled, skewed, etc.
- Video: played backward/forward, upside down, etc.
- Some transformations preserve the relationships you want to learn, but force the network to learn to form stronger invariant representations
  - Access to a wide range of experiences strengthens invariant representations and thus generalization
  - Think about your data and what augmentations might allow this to happen…

Combine Techniques

In the end, you are deciding on a strategy which may have several parts (weight-decay, noise, dropout, augmentation, etc.)

Mixing and matching different methods is common and results in unique learning properties that may (or may not) improve generalization…

A good first try is to see if some **data augmentation** helps, then **dropout**, and then maybe noise, etc.

Training may take *longer* but overall result in **better generalization** when using these techniques