CSC413 Neural Networks and Deep Learning

Lecture 6: Generalization

February 13/15, 2024

Table of Contents

1. From Optimization to Generalization
2. Training Expressive Models yet Avoiding Overfitting
3. Bias-Variance Tradeoff

Lecture Plan

- Generalization
- Strategies to mitigate overfitting
- Reminder on bias/variance decomposition

Section 1

From Optimization to Generalization

From Optimization to Generalization

- Recall that in ML, the average cost function that we minimize is defined over the training data.

$$\mathcal{E}(w) = \frac{1}{N} \sum_{i \in \text{training}} \mathcal{L}(f_w(x^{(i)}), t^{(i)})$$

- From optimization perspective, having a good optimizer means that the result is close to the minimum of the average loss function w.r.t. the training data.

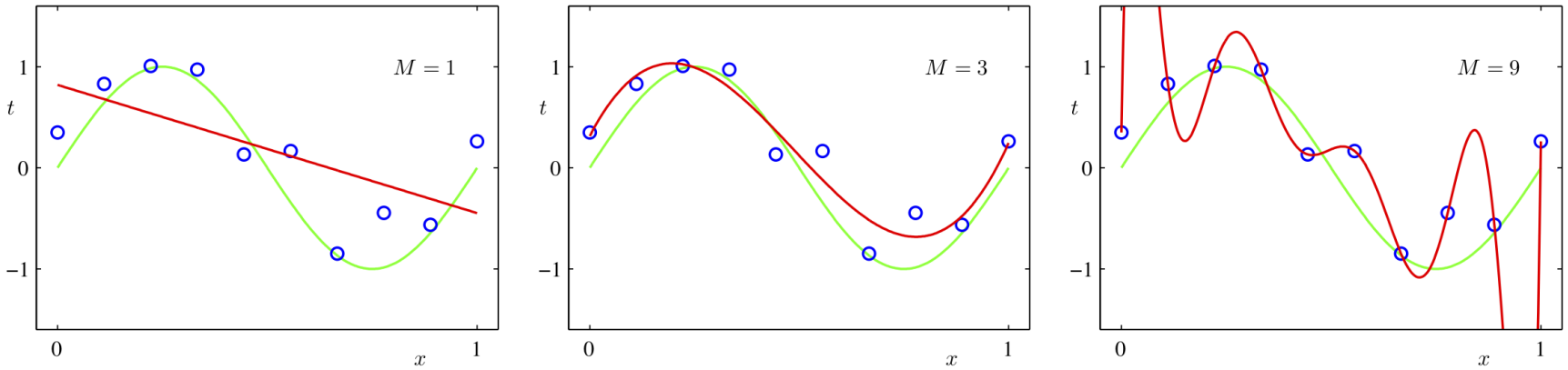
$$\min_w \mathcal{E}(w)$$

- The ultimate goal in ML, however, is different: we want to **generalize** well on data that we have not seen before.
  - Memorization is not enough!
- Merely optimizing the training loss is not all that we care about.

Some Important Questions

- How do we know how well a model will perform on new data?
- How do we choose between different neural network models?
  - Different number of hidden units, number of layers, etc.
- How can we make sure our optimizer finds a solution that performs well on new data?

Generalization – Overfitting and Underfitting



We'd like to minimize the generalization error, that is, the error on novel (new) examples.

Overfitting and Underfitting

Underfitting:

- The model is simple and doesn't fit the data
- The model does not capture *discriminative* features of the data

Overfitting:

- The model is too complex and does not generalize
- The model captures information about patterns in training set that happened by chance
  - e.g. Ringo happens to be always wearing a red shirt in the training set
  - Model learns: high red pixel content $=>$ predict Ringo

The Training Set

The training set is used

- to determine the value of the **parameters**

The model's prediction accuracy over the training set is called the **training accuracy**.

Q: Can we use the **training accuracy** to estimate how well a model will perform on new data?

The Training Set

The training set is used

- to determine the value of the **parameters**

The model's prediction accuracy over the training set is called the **training accuracy**.

Q: Can we use the **training accuracy** to estimate how well a model will perform on new data?

- No! It is possible for a model to fit well to the training set, but fail to *generalize*
- We want to know how well the model performs on *new data* that we didn't already use to optimize the model

The Test Set

We set aside a **test set** of labelled examples.

The model's prediction accuracy over the test set is called the **test accuracy**.

The purpose of the test set is to give us a good estimate of how well a model will perform on new data.

Q: In general, will the test accuracy be *higher* or *lower* than the training accuracy?

Model Selection

How to make decisions such as

- the number of layers?
- the number of units in each layer?
- the type of non-linear activation?

Q: Why can't we use the test set to determine which model we should deploy?

Model Selection

How to make decisions such as

- the number of layers?
- the number of units in each layer?
- the type of non-linear activation?

Q: Why can't we use the test set to determine which model we should deploy?

- If we use the test set to make modeling decisions, then we will overestimate how well our model will perform on new data!
- We are "cheating" by "looking at the test"

The Validation set

We need a third set of labeled data called the **validation set**.

The model's prediction accuracy over the validation set is called the **validation accuracy**.

This dataset is used to:

- Make decisions about the aspects of the model that are **not differentiable** and cannot easily be optimized via gradient descent
- Example: choose the number of layers, units in each layer, learning rate, etc.
  - These model settings are called **hyperparameters**
- The validation set is used to optimize **hyperparameters**

Splitting the data set

Example split:

- 60% Training
- 20% Validation
- 20% Test

The actual split depends on the amount of data that you have.

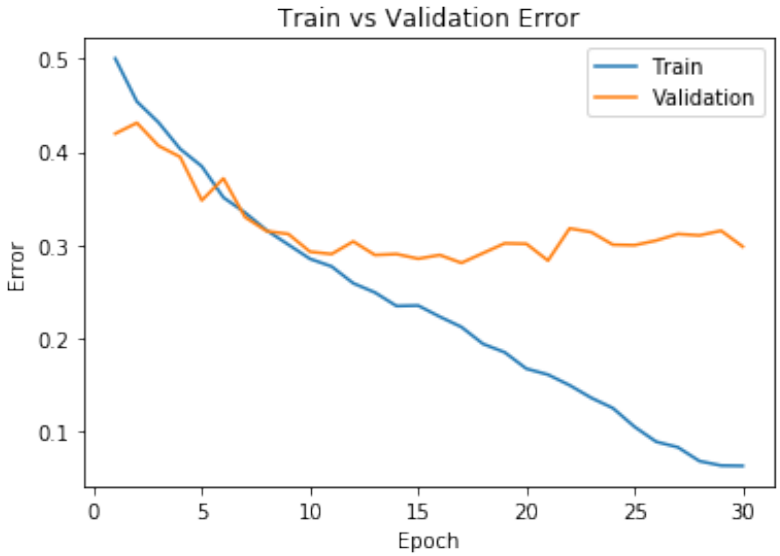
If you have more data, you can get a way with a smaller percentage of data dedicated to the validation set.

- Why?

Detecting Overfitting

Learning curve:

- **x-axis**: epochs or iterations
- **y-axis**: cost, error, or accuracy

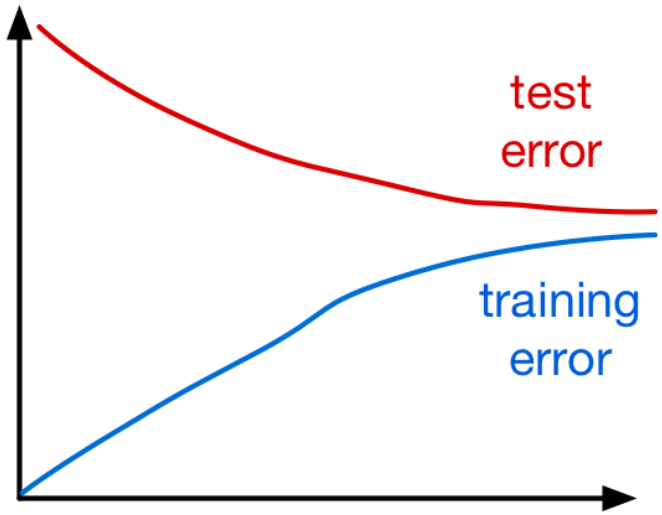


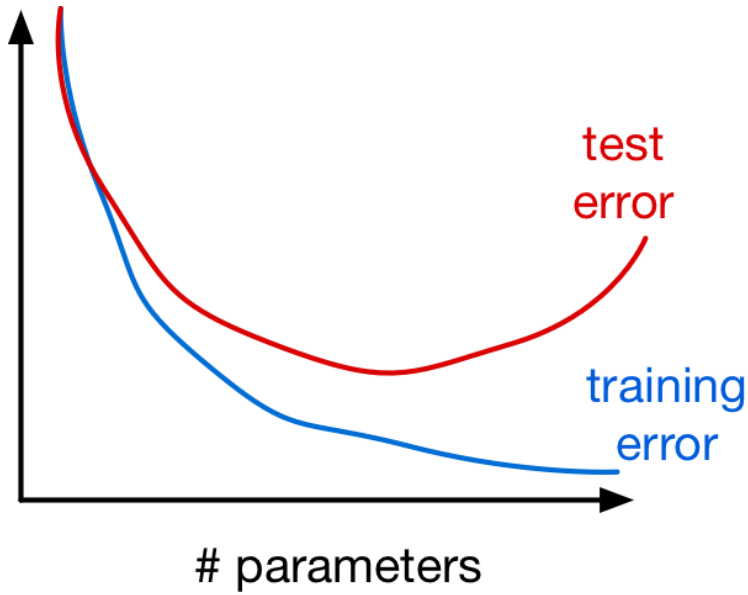
Q: In which epochs is the model overfitting? Underfitting?

Q: Why don't we plot the test accuracy plot?

Effect of Training Data Size and Model Complexity

Recall from the Intro to ML course that the effect of # training examples and # parameters on Training and Test errors look like this:





- The second figure may not be completely accurate for a large NN.
  - Even though larger NN have more parameters, they may have "nicer" optimization landscape that leads to solutions that generalize better.
  - Understanding this is an active area of research.

Section 2

Training Expressive Models yet Avoiding Overfitting

Training Expressive Models yet Avoiding Overfitting

Two forces in action:

- We would like our models to be expressive enough that they can capture the structure in data
  - Higher-order polynomial features for a linear model, larger decision trees, etc.
  - Deeper and wider NNs
- We would like our models to avoid overfitting
  - We only want to capture the "structure" while avoiding learning the "noise".

Q: How did we achieve this in our Intro to ML course?

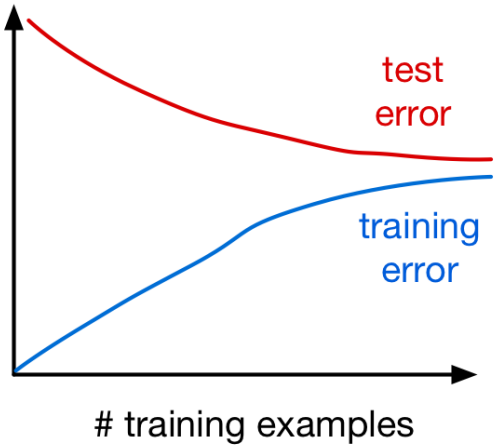
Strategies to Prevent Overfitting

- Data Augmentation
- Reducing the number of parameters
- Weight decay (or regularization/penalization)
- Early stopping
- Ensembles
- Stochastic regularization (e.g. dropout)

The best-performing models on most benchmarks use some or all of these tricks.

Let us take a look at them closely!

Strategy #1: Data Augmentation



The best way to improve generalization is to collect more data!

But if we have already collected all the data that we could, we can augment the training data by *transforming* the examples. This is called **data augmentation**.

Some examples of data augmentation for images:

- translation
- horizontal or vertical flip
- rotation

Data Augmentation

- We should only warp the training examples, not the validation or test examples. (why?)
- The choice of transformations depends on the task.
  - Horizontal flip for object recognition, but not handwritten digit recognition. (why?)

Q: Can you think of data augmentation for audio? for text?

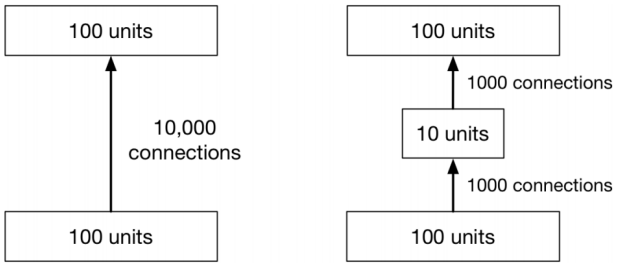
Strategy #2: Reducing the Number of Parameters

Networks with fewer trainable parameters *may be* less likely to overfit.

- But recall the comment earlier about the optimization landscape!

We can reduce the number of layers, or the number of parameters per layer.

Adding a **bottleneck layer** is another way to reduce the number of parameters



The first network is strictly more expressive than the second (i.e., it can represent a strictly larger class of functions). (Why?)

Remember how linear layers don't make a network more expressive? They might still improve generalization.

Strategy #3: Weight Decay (or Regularization)

Idea: Regularize/penalize **large weights** by adding a term (e.g. $\sum_k w_k^2$) to the cost function.

- This encourages the weights to be small in magnitude:

$$\mathcal{J}_{\text{reg}} = \mathcal{J} + \lambda \mathcal{R} = \mathcal{J} + \frac{\lambda}{2} \sum_j w_j^2$$

- The gradient descent update can be interpreted as **weight decay**:

$$w \leftarrow w - \alpha \left(\frac{\partial \mathcal{J}}{\partial w} + \lambda \frac{\partial \mathcal{R}}{\partial w} \right)$$

$$= w - \alpha \left(\frac{\partial \mathcal{J}}{\partial w} + \lambda w \right)$$

$$= (1 - \alpha\lambda) w - \alpha \frac{\partial \mathcal{J}}{\partial w}$$

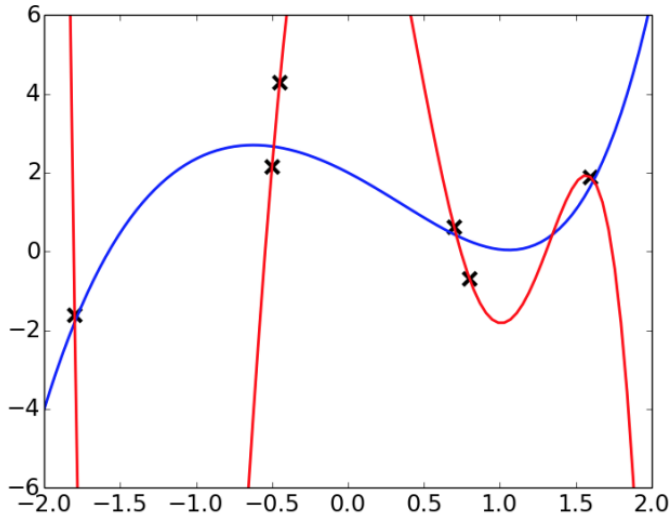
Weight Decay

Why is it not desirable to have very large weights?

Because large weights mean that the prediction relies **a lot** on the content of one feature (e.g. one pixel)

Small vs. Large Weights: Example 1

The red polynomial overfits. Notice it has really large coefficients



$$y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2$$

$$y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12$$

Small vs. Large Weights: Example 2

- Suppose inputs x_1 and x_2 are nearly identical. The following two networks make nearly the same predictions:

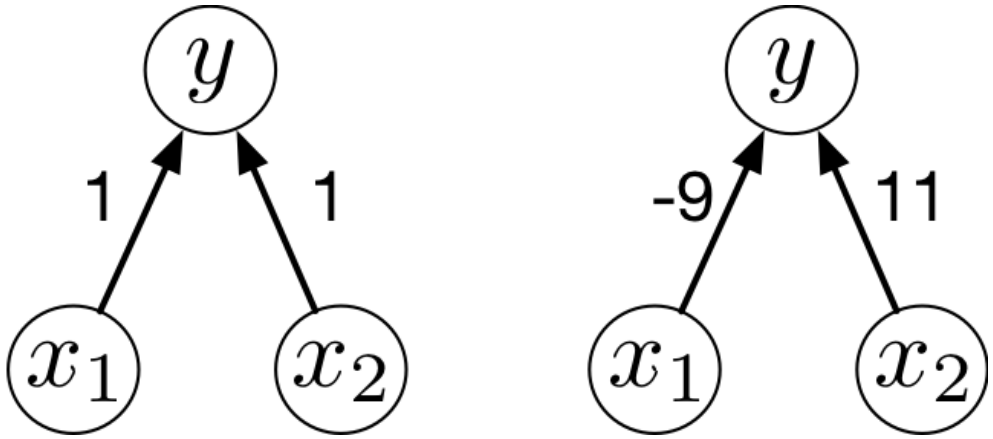


Figure 1: Output sensitivity to large weights

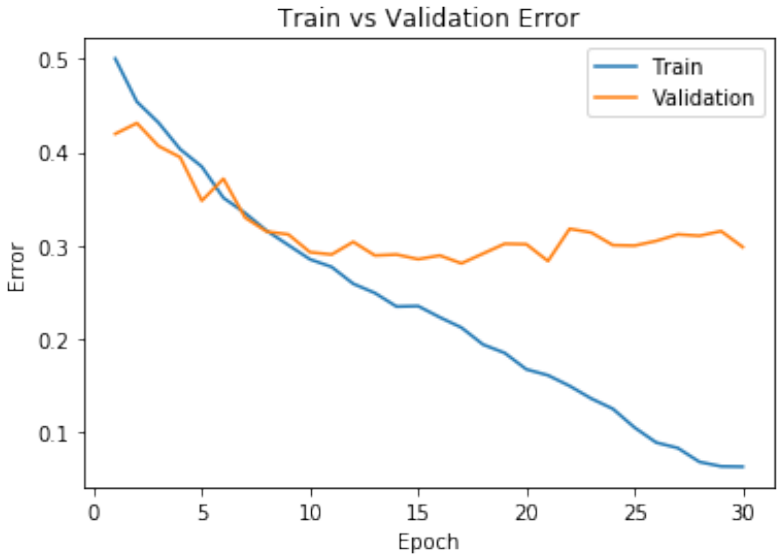
- But the second network might make weird predictions if the test distribution is slightly different (e.g., x_1 and x_2 match less closely).

Many Choices of Regularizers

- L^1 regularization: add a term $\sum_{j=1}^{D} |w_j|$ to the cost function
  - Mathematically, this term encourages weights to be exactly 0
- L^2 regularization: add a term $\sum_{j=1}^{D} w_j^2$ to the cost function
  - Mathematically, in each iteration the weight is pushed towards 0
- Combination of L^1 and L^2 regularization: add a term $\sum_{j=1}^{D} [|w_j| + w_j^2]$ to the cost function

Strategy #4: Early Stopping

Idea: Stop training when the validation error starts going up.



In practice, this is implemented by **checkpointing** (saving) the neural network weights every few iterations/epochs during training.

We choose the checkpoint with the best validation error to actually use. (And if there is a tie, use the **earlier** checkpoint)

Why does early stopping work?

Weights start off small, so it takes time for them to grow large.

Therefore, stopping early has a similar effect to weight decay.

If you're using sigmoid units, and the weights start out small, then the inputs to the activation functions take only a small range of values.

- The neural network starts out approximately linear, and gradually becomes non-linear (and thus more powerful)

Strategy #5: Ensembles

If a loss function is convex (with respect to the predictions), you have a bunch of predictions for an input, and you don't know which one is best, you are always better off averaging them!

$$\mathcal{L}(\lambda_1 y_1 + \ldots \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \ldots \lambda_N \mathcal{L}(y_N, t)$$

for $\lambda_i \geq 0$ and $\sum_i \lambda_i = 1$

Idea: Build multiple candidate models, and average the predictions on the test data.

This set of models is called an **ensemble**.

Examples of Ensembles

- Train neural networks starting from different random initialization (might not give enough diversity)
- Train different network on different subset of the training data (called **bagging**)
- Train networks with different architectures, hyperparameters, or use other machine learning models

Ensembles can improve generalization substantially.

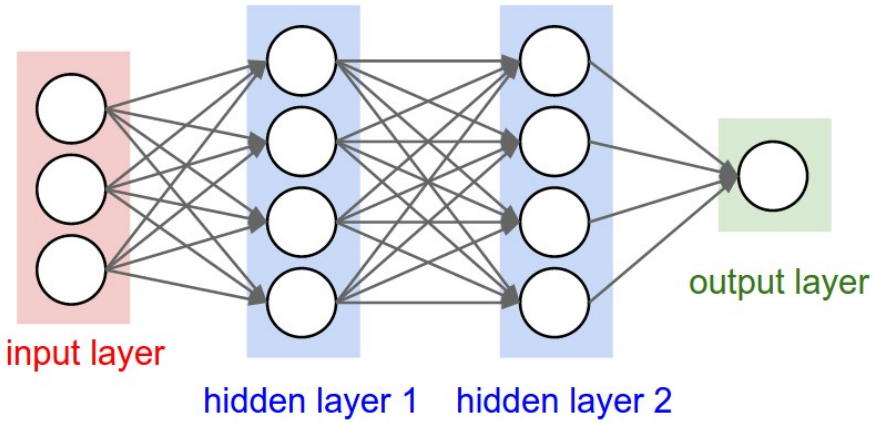
However, ensembles are expensive.

Strategy #6: Stochastic Regularization

For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as **stochastic regularization**.

One example is **dropout**: in each training iteration, randomly choose a portion of **activations** to set to 0.

The probability p that an activation is set to 0 is a hyperparameter.



Stochastic Regularization

More mathematically:

$$h_j = \begin{cases} \phi(z_j) & \text{with probability } 1 - \rho \\ 0 & \text{with probability } \rho, \end{cases}$$

or equivalently,

$$h_j = m_j \phi(z_j)$$

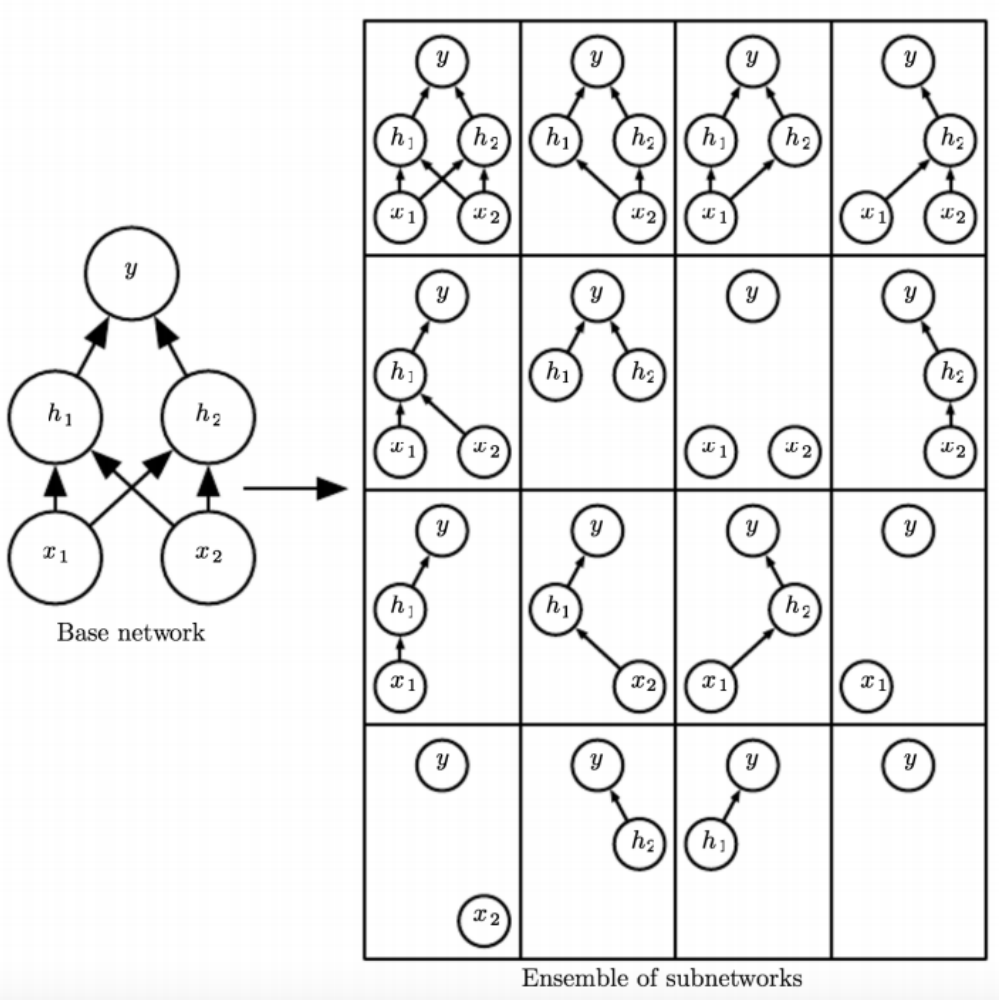
with m_j being a Bernoulli random variable, independent for each unit.

Backpropagation:

$$\bar{z}_j = \bar{h}_j m_j \phi'(z_j)$$

Dropout

Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units)



Dropout at Test Time

- Most principled thing to do: run the network lots of times independently with different dropout masks, and average the predictions.
  - Individual predictions are stochastic and may have high variance, but the averaging fixes this.
- In practice: don't do dropout at test time, but multiply the weights by $1 - \rho$
  - Since the weights are ON for $(1 - \rho)$-fraction of the time, this matches their expected value of their activation value.

Stochastic Regularization

- Dropout can help performance quite a bit, even if you are already using weight decay.
- Other stochastic regularizers have been proposed:
  - The stochasticity in SGD updates has been observed to act as a regularizer, helping generalization.
    - Increasing the mini-batch size may improve training error at the expense of test error!
  - Batch normalization (mentioned last week for its optimization benefits) also introduces stochasticity, thereby acting as a stochastic regularizer.

Section 3

Bias-Variance Tradeoff

Bias-Variance Tradeoff

- We can understanding ML algorithms better in terms of their bias and variance (also closely related to approximation and estimation errors).
- Our discussion here is brief and is only a reminder.
- Consult your Intro to ML course for more detail.
  - Example: Bias and Variance lecture, CSC2515 – Fall 2022

Expected Test Error for Regression

- Training set $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ drawn i.i.d. from distribution $P(X, Y)$. Let's write this as $D \sim P^n$.
- Assume for simplicity this is a regression problem with $y \in \mathbb{R}$ and L_2 loss.
- What is the expected test error for a function $h_D(x) = y$ trained on the training set $D \sim P^n$, assuming a learning algorithm $\mathcal{A}$? It is:

$$\mathbb{E}_{D \sim P^n, (x,y) \sim P} \left[(h_D(x) - y)^2 \right]$$

- The expectation is taken with respect to possible training sets $D \sim P^n$ and the test distribution P. Let's write the expectation as $\mathbb{E}_{D,x,y}$ for notational simplicity.
- Note that this is the *expected test error* not the *empirical test error* that we report after training. How are they different?

Decomposing the Expected Test Error

Let's start by adding and subtracting the same quantity

$$\mathbb{E}_{D,x,y}\left[(h_D(x) - y)^2\right] = \mathbb{E}_{D,x,y}\left[(h_D(x) - \hat{h}(x) + \hat{h}(x) - y)^2\right]$$

Denote

- $\hat{h}(x) = \mathbb{E}_{D\sim P^n}[h_D(x)]$ is the expected regressor over possible training sets, given the learning algorithm $\mathcal{A}$.
- $\hat{y}(x) = \mathbb{E}_{y|x}[y]$ is the expected label given x. Labels might not be deterministic given x.

Decomposing the Expected Test Error

After some algebra*, we can show that:

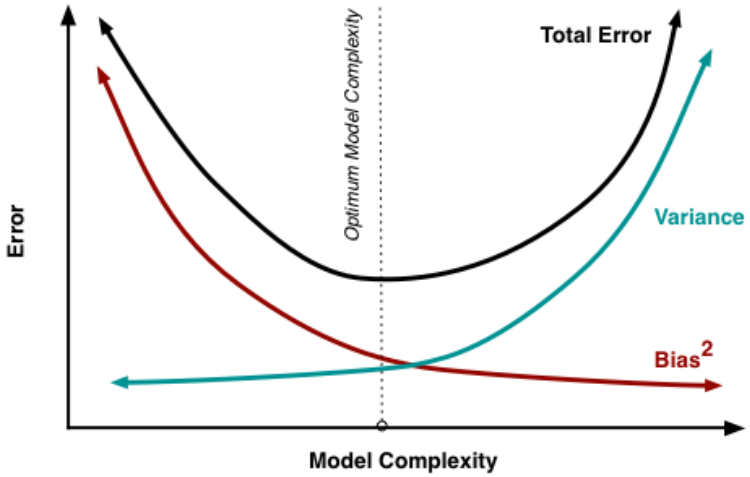
$$\underbrace{\mathbb{E}_{D,x,y}\left[(h_D(x)-y)^2\right]}_{\text{Expected test error}} = \underbrace{\mathbb{E}_{D,x}\left[(h_D(x)-\hat{h}(x))^2\right]}_{\text{Variance}} + \underbrace{\mathbb{E}_{x,y}\left[(\hat{y}(x)-y)^2\right]}_{\text{Noise}} + \underbrace{\mathbb{E}_{x}\left[(\hat{h}(x)-\hat{y}(x))^2\right]}_{\text{Bias}}$$

- You can find the proof in this URL (*)

Bias, Variance, and Noise

- **Variance**: Captures how much your regressor h_D changes if you train on a different training set. How "over-specialized" is your regressor h_D to a particular training set D? I.e. how much does it overfit? If we have the best possible model for our training data, how far off are we from the average regressor $\hat{h}$?
- **Bias**: What is the inherent error that you obtain from your regressor h_D even with infinite training data? This is due to your model being "biased" to a particular kind of solution (e.g. linear model). In other words, bias is inherent to your model/architecture.
- **Noise**: How big is the data-intrinsic noise? This error measures ambiguity due to your data distribution and feature representation. You can never beat this, it is an aspect of the physical data generation process, over which you have no control. You cannot improve this with more training data. It is sometimes called "aleatoric uncertainty".

The Bias-Variance Tradeoff



- If you use a high-capacity model, you will get low bias, but the variance over different training sets will be high.
- If you use a low-capacity model, you will get high bias, but the variance over different training sets will be low.
- There is a sweet spot that trades off between the two.