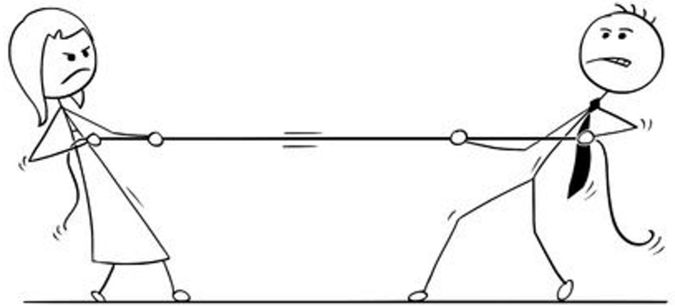
Overfitting, underfitting and regularization (Bias-variance tradeoff)

Underfitting (high-bias)

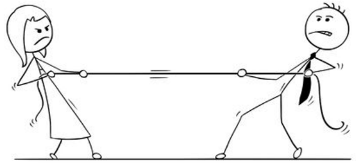


Overfitting (high-variance)

Overfitting (variance) and underfitting (bias)

“The bias-variance tradeoff is a central problem in supervised learning”

Underfitting (high-bias)



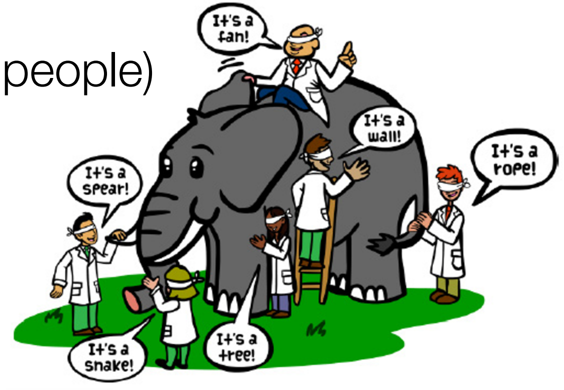
Overfitting (high-variance)

High bias is too less optimization (having too few blind people)

- Model misses to learn many representative training data

High variance is too much optimization (having too many blind people)

- Model learns unrepresentative training data as well



We do not want either!!

We want generalization:

- A few but ‘smart’ blind person “One side is snake like and the other side rope like” or “...”, etc.

How to prevent overfitting?

We fight overfitting using “regularization” techniques:

1. Get more training data (more real data)
2. Data augmentation (more fake data)
3. Reduce the network’s size
   - Number of neurons and layers
4. Add weight regularization
   - L1 and L2 penalties
5. Adding dropout layers
6. Early stopping and model checkpointing
7. Adding batch normalization layers (not fully considered a regularization technique)

3. Regularization by reducing the network’s size

- Model with more parameters = More memorization capacity
  - More capacity allows a model to easily learn a perfect dictionary-like mapping between ‘training samples’ and their ‘targets’
    - a mapping ‘without any generalization power’
- The challenge is generalization, not fitting
  - Deep learning models tend to be good at fitting to the training data
- “Too much capacity vs not enough capacity”
  - If the network has limited memorization resources, it won’t be able to learn this mapping as easily
  - The model will starve for memorization resources
- So, what is the magical formula?
  - We must evaluate an array of different architectures (on your validation set, not on test set) in order to find the correct model size for your data
  - The general workflow to find an appropriate model size:
    - Start with relatively few layers and parameters, and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss

How do I know if my model is already too large?

Listing 4.3 Original model

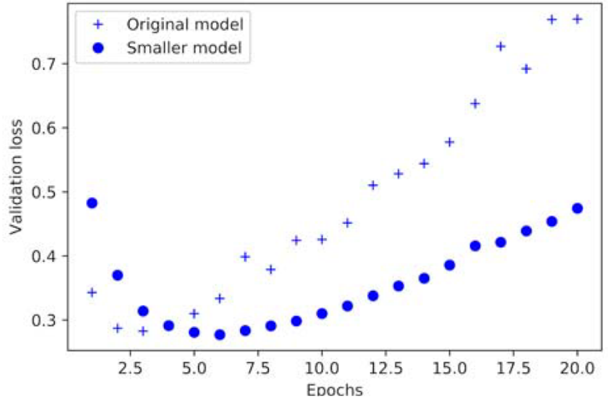
```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

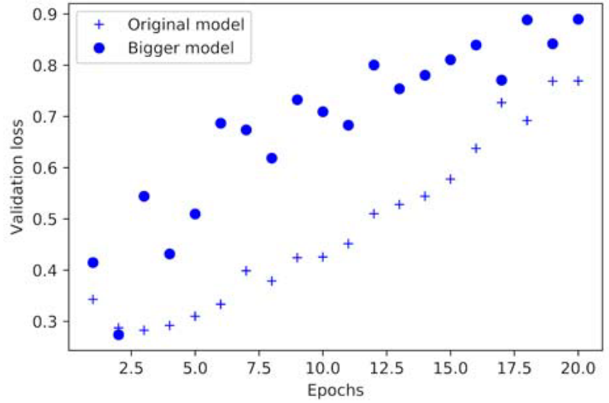
Listing 4.4 Version of the model with lower capacity

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Listing 4.5 Version of the model with higher capacity

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```





Which network nears zero loss more quickly?

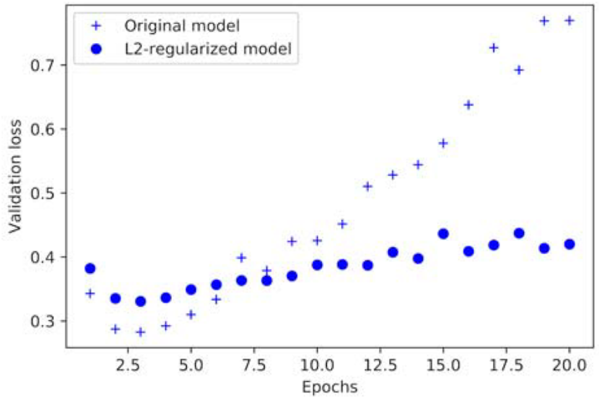
4. Adding weight regularization

- Occam's razor: Given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions
- Simpler models are less likely to overfit than complex ones
    - A simple model in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters)
- How to reduce the complexity of a network (without changing # of parameters?
    - Force its weights to take only small values
    - This will make the distribution of weight values more **regular**
- Weight regularization
    - Adding to the loss function of the network - "a cost associated with having large weights"
    - I.e. don't let the weight values grow

Two ways to regularize weights: L_1 and L_2

1. L_1 regularization
   - The cost added is proportional to the absolute value of the weight coefficients
2. L_2 regularization
   - The cost added is proportional to the square of the value of the weight coefficients
   - L_2 regularization is also called weight decay in the context of neural networks

```
from tensorflow.keras import regularizers
…
model.add(Conv2D(4,(3,3), kernel_regularizer=regularizers.l1, ...))
model.add(Flatten())
model.add(Dense(10, kernel_regularizer=regularizers.l2(0.001), ...))
...
```



What does `l2(0.001)` do?

Every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value` to the total loss of the network.

5. Adding dropout

- Dropout is one of the most effective and most commonly used regularization
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training
  - Let's say a given layer would normally return a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training
  - After applying dropout, this vector will have a few zero entries distributed at random
    - For example, `[0, 0.5, 1.3, 0, 1.1]`
- The **dropout rate** is the fraction of the features that are zeroed out
  - It's usually set between 0.2 and 0.5

```
…
model.add(Conv2D(16, 3, activation = 'sigmoid', ...)
model.add(Dropout(rate = 0.25))
model.add( Conv2D( 4, ( 3, 3 ), activation = 'relu' ) )
…
```

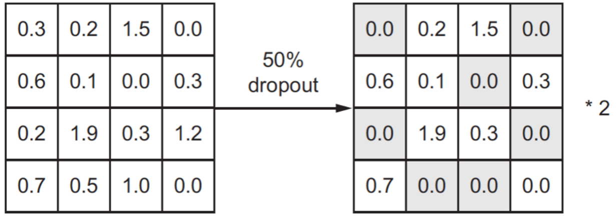
How is dropout handled during testing?

Strategy 1: At test time, no units are dropped out

- Instead, the layer's output values are scaled down by a factor equal to the dropout rate
- `layer_output *= 0.5`
- This is to balance for the fact that more units are active than at training time

Strategy 2: Inverted dropout

- `activation = activation / keep_probability` to scale up during the training



0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

Why does dropout work?

- Hinton was inspired by a fraud-prevention mechanism used by banks
    - “I went to my bank. The tellers kept changing and I asked one of them why. He said he didn’t know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.”
- The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren’t significant

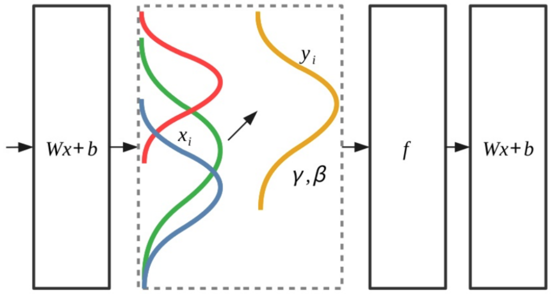
7. Batch normalization

Problem: The distribution of each layer's input changes during training

Solution: Fix the distribution

Batch normalization is a key solution to avoid/minimize the vanishing gradient problem

Ensure the output statistics of a layer are fixed.



```
model.add( Conv2D( 16, ( 3, 3 ), activation = 'sigmoid', input_shape = xtrain[0, :, :, :].shape ) )
model.add(BatchNormalization())
model.add( Conv2D( 4, ( 3, 3 ), activation = 'relu' ) )
```

Example

```
..AUGMENTED INPUT..

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, BatchNormalization, MaxPool2D, Dropout
model = Sequential()
model.add( Conv2D( 64, ( 3, 3 ), activation = 'sigmoid', input_shape = xtrain[0, :, :, :].shape ) )
model.add(BatchNormalization())
model.add( Conv2D( 64, ( 3, 3 ), activation = 'relu' ) )
model.add(MaxPool2D(2, 2))
model.add( Conv2D( 64, ( 3, 3 ), activation = 'relu' ) )
model.add(Dropout(0.25))
model.add( Conv2D( 64, ( 3, 3 ), activation = 'relu' ) )
model.add(BatchNormalization())
model.add( Conv2D( 64, ( 3, 3 ), activation = 'relu' ) )
model.add(BatchNormalization())
model.add( Conv2D( 64, ( 3, 3 ), activation = 'relu' ) )
model.add(BatchNormalization())
model.add( Flatten() )
model.add( Dense( 10, activation = 'relu' ) )
model.add( Dense( 10, activation = 'softmax' ) )
model.summary()


..TRAINING WITH CALLBACKS..
```

Training a very deep ConvNet: The villains

```
model = Sequential()
model.add(Conv2D(filters = 16, kernel_size = 5, activation = 'relu', input_shape = (28,28,1)))

for i in range(1000):
    model.add(Conv2D(filters = 4, kernel_size = 5, activation = 'relu', padding='same'))

model.add(Conv2D(filters = 16, kernel_size = 5, activation = 'relu'))
model.add(MaxPooling2D(pool_size = 2, strides = 2))
model.add(Flatten())
model.add(Dense(units = 16, activation = 'relu'))
model.add(Dense(units = 16, activation = 'relu'))
model.add(Dense(units = 10, activation = 'softmax'))
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])

print(model.summary())
```

Some major villains when training 'very deep' networks:

- Vanishing Gradient, Exploding Gradient, and Internal Covariate Shift

'ReLU' and 'BatchNormalization' greatly resolve this

- But, BatchNormalization slows down training