BST 261: Data Science II

Lecture 6

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

April 4, 2018

Regularization

Regularization

- One of the easiest ways to fight overfitting (when more training data isn't available) is with regularization: any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.
- In the context of deep learning, most regularization strategies are based on regularizing estimators.
- Regularization of an estimator works by trading increased bias for reduced variance.
- An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.
- Controlling the complexity of the model is not a simple matter of finding the model of the right size, with the right number of parameters. Instead, we might find -and indeed in practical deep learning scenarios, we almost always do find - that the best fitting model (in the sense of minimizing generalization error) is a large model that has been **regularized appropriately**.
- We will focus on 3 different methods:
  - Reducing the size of the network
  - Weight regularization
  - Dropout

Reducing Network Size

Reducing network size

- The simplest way to prevent overfitting is to reduce the network's size: the number of learnable parameters in the model
- This is determined by the number of layers and units in each of those layers
- The number learnable parameters is what we referred to before as a model's 'capacity'
- As we saw before, we need to find a 'just right' solution that doesn't under or overfit
- There isn't a formula for this - you must train several different networks and evaluate them all
- Start with relatively few layers and units, and work your way up until you see diminishing returns in terms of validation loss

IMDB Classification Example

- Original Model:

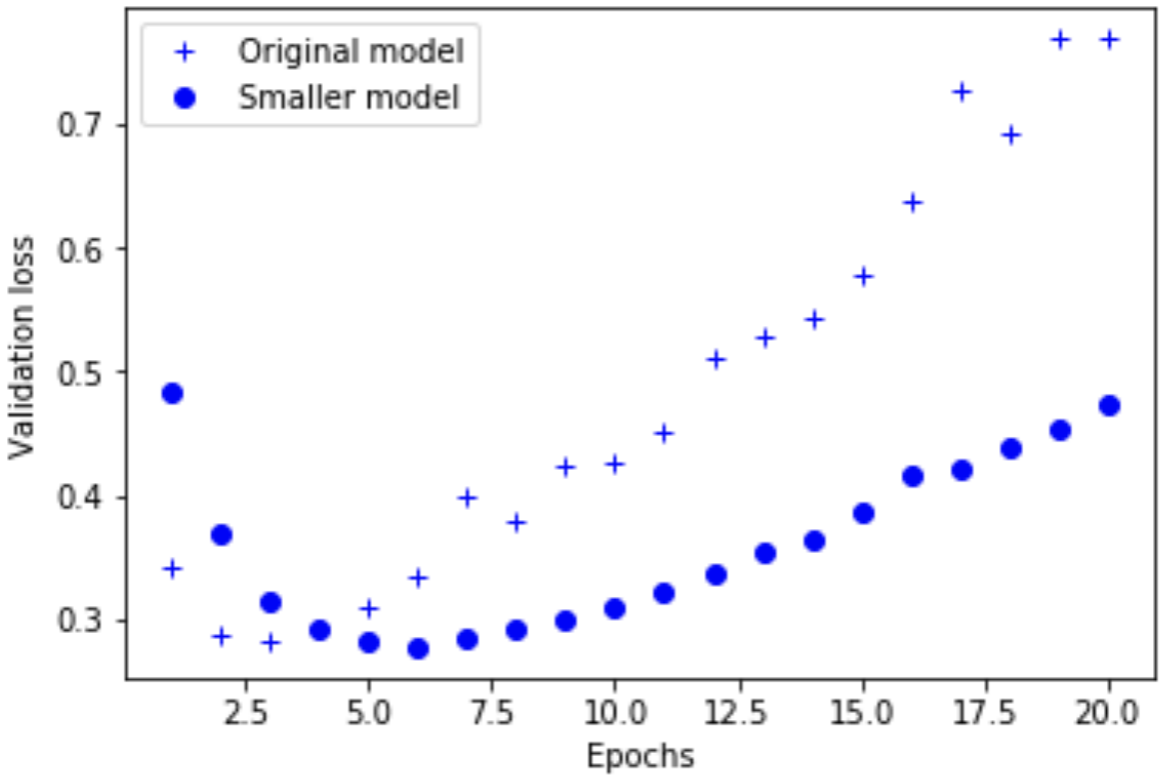
```
from keras import models
from keras import layers

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'))
```

- Lower Capacity Model:

```
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'))
```

IMDB Classification Example

- The smaller network starts overfitting later than the original network, and its performance degrades more slowly once it starts overfitting



IMDB Classification Example

- Original Model:

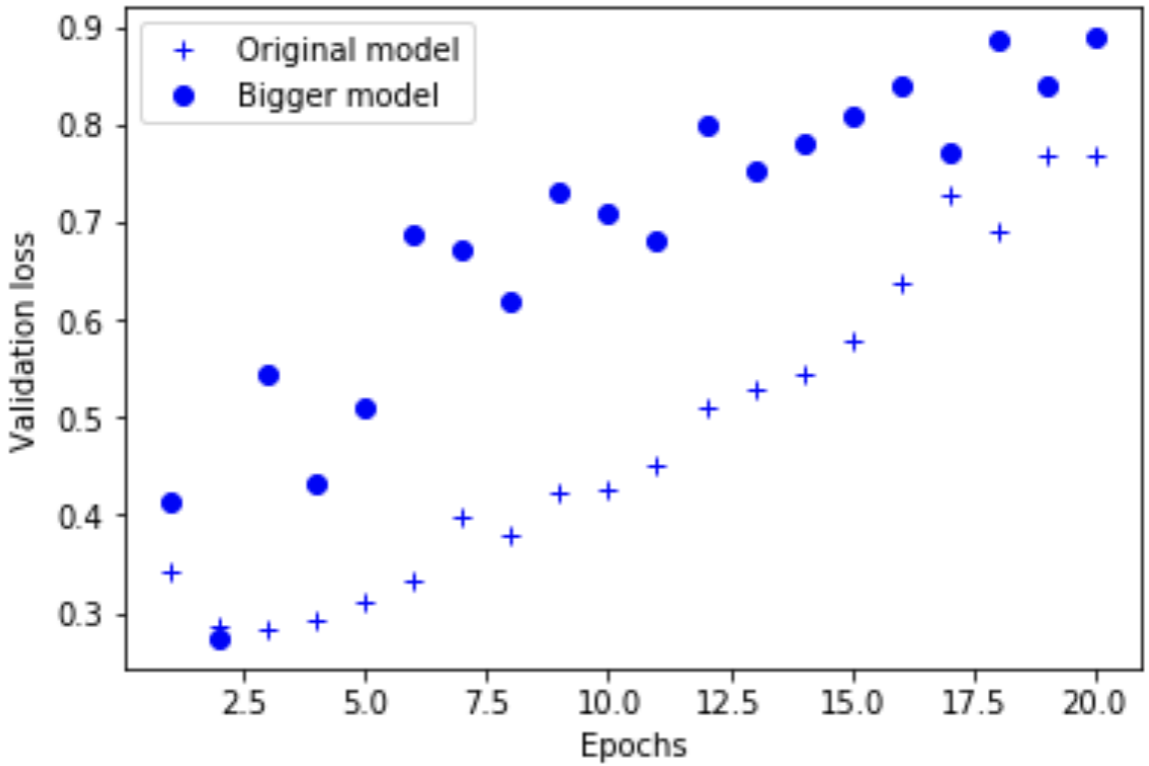
```
from keras import models
from keras import layers

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'))
```

- Higher Capacity Model:

```
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'))
```

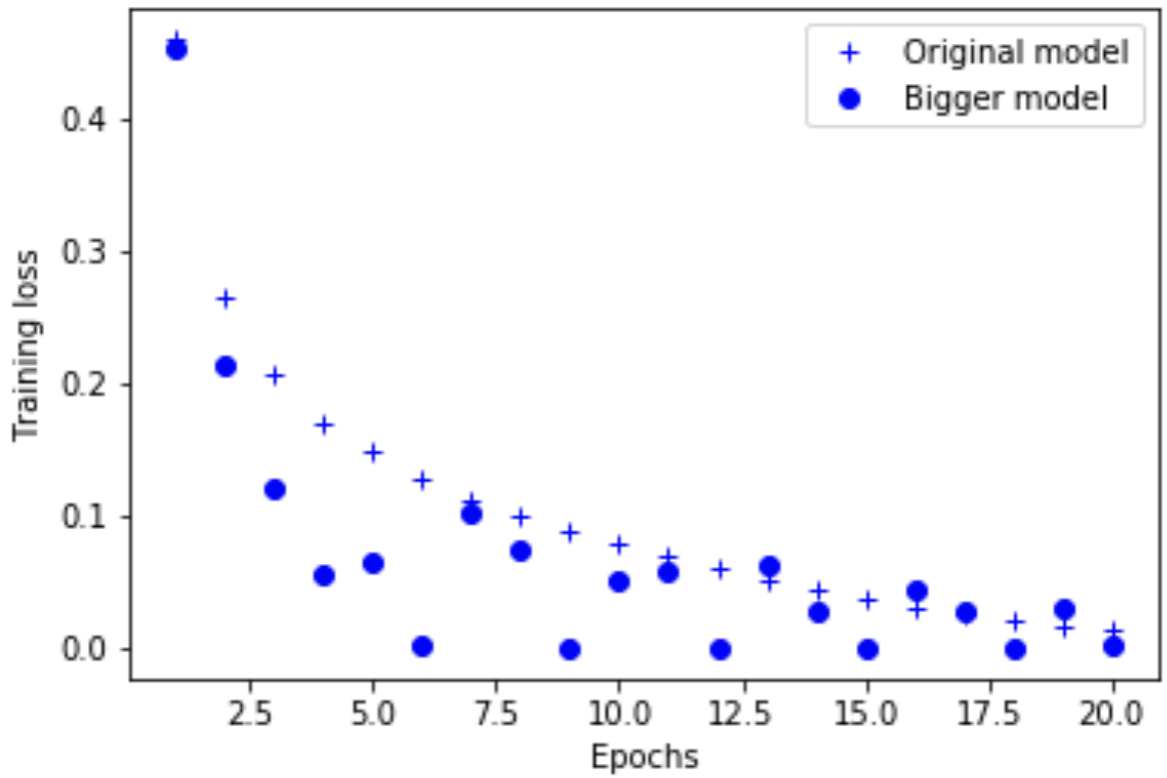
IMDB Classification Example

- The bigger network starts overfitting almost immediately and it overfits much more severely
- The bigger network's validation loss is also noisier



IMDB Classification Example

- The bigger network decreases its training error to 0 very quickly since the more capacity a model has the quicker it can model the training data
- However, this performance on the training data makes it more susceptible to overfitting



Weight Regularization

Weight Regularization

- Here we will limit the capacity of the model by adding a parameter norm penalty $\Omega(\boldsymbol{w})$ to the cost function $J(\boldsymbol{w}, b)$:

$$\tilde{J}(\boldsymbol{w}, b) = J(\boldsymbol{w}, b) + \lambda\Omega(\boldsymbol{w}) \tag{1}$$

- $\lambda \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term Ω
- Setting λ to 0 results in no regularization and larger values of λ correspond to more regulation
- Note that we are only penalizing the weights at each layer and not the biases (b)
- The biases typically require less data than the weights to fit accurately, and regularizing the bias parameters can introduce a significant amount of overfitting
- In the context of neural networks, it can be desirable to use a separate penalty with a different λ coefficient for each layer. However, this can be very computationally expensive due to having to search for the correct value of multiple hyperparameters

Weight Regularization

- The simplest and most common kinds of parameter norm penalty is L^2 regularization
- L^2 **regularization**: The cost added is proportional to the *square of the value* of the weight coefficients (the L^2 norm of the weights)
- Also called **weight decay** in the context of neural networks
- Drives the weights closer to the origin
- Also known as **ridge regression** or **Tikhonov regularization** in other academic communities
- The L^2 regularization term is:

$$\Omega(\boldsymbol{w}) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 = \frac{1}{2} \sum_{j=1}^{n} \boldsymbol{w}_j^2 = \frac{1}{2} \boldsymbol{w}^T \boldsymbol{w} \tag{2}$$

Weight Regularization

- The corresponding cost function is:

$$\tilde{J}(\boldsymbol{w}, b) = J(\boldsymbol{w}, b) + \lambda\Omega(\boldsymbol{w}) = J(\boldsymbol{w}, b) + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w} \tag{3}$$

- With a corresponding gradient of:

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}, b) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w}, b) + \lambda\boldsymbol{w} \tag{4}$$

- To take a single gradient step to update the weights, we perform the following update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\lambda\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}, b)) \tag{5}$$

- Written another way:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\lambda)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w}, b) \tag{6}$$

- The addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update
- Weights that do not contribute to reducing the cost function are decayed away and driven closer to the origin

Weight Regularization

- A less commonly used, but still popular form of weight decay is L^1 regularization
- L^1 **regularization**: The cost added is proportional to the *absolute value* of the weight coefficients (the L^1 norm of the weights)

$$\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 = \sum_{j=1}^{n} |\boldsymbol{w}_j| \tag{7}$$

- Rather than a linear scaling of the gradient, this form is a constant scaling
- Results in a solution that is more **sparse** (more weights set to 0)
- This sparsity property has been used extensively as a feature selection mechanism
- In the context of a linear model and least-squares cost function, L^1 regularization is called **LASSO**

Regularization in Keras

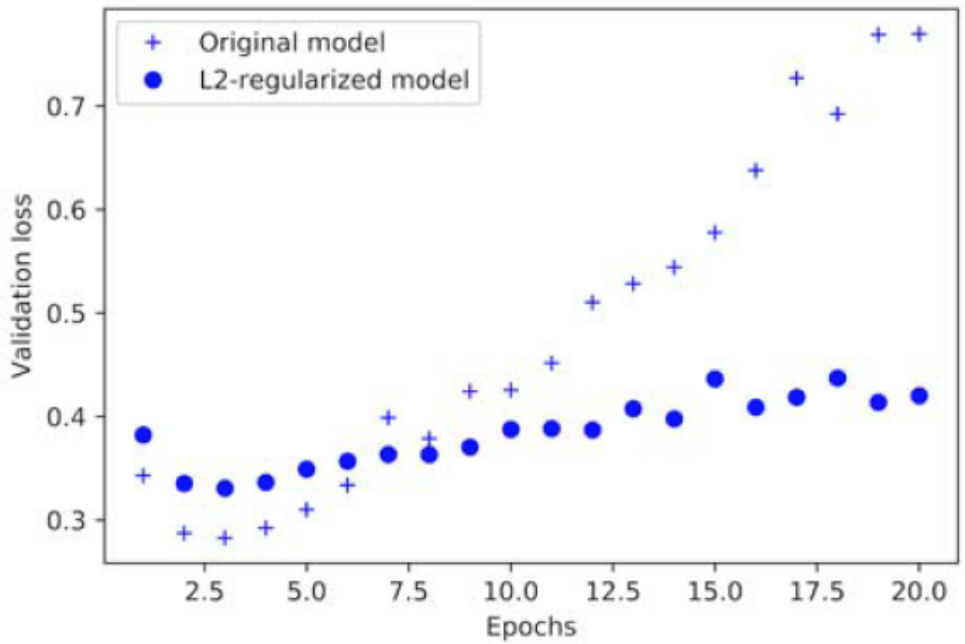
- In Keras, weight regularization is added by passing ***weight regularizer instances*** to layers as keyword arguments. Let's add L^2 weight regularization to the IMDB classification network:

```
from keras import regularizers
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- `l2(0.001)` means every coefficient in the weight matrix of the layer will add `0.001 *weight_coefficient_value` to the total loss of the network. Note that because this penalty is only added at training time, the loss for this network will be much higher at training than at test time.

Regularization in Keras

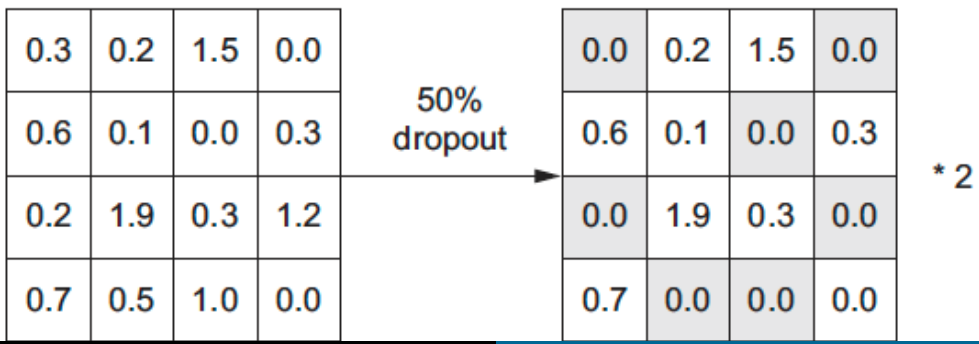
- After weight regularization, the model with L^2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.



Dropout

Dropout

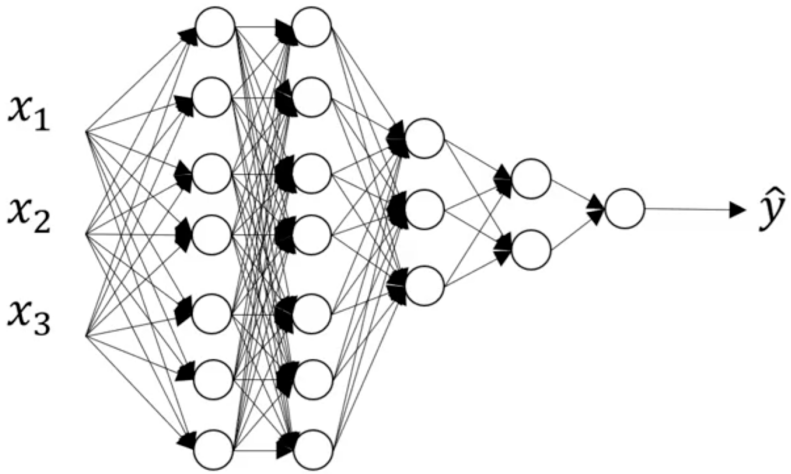
- One of the most effective and most commonly used regularization techniques for neural networks
- Provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- It consists of randomly *dropping out* (setting to 0) a number of nonoutput units.
- For example, let's say a given layer would normally return a vector $[0.1, 0.6, 0.9, 0.3, 0.15]$ for a given input during training. After applying dropout, this vector will have a few 0 entries distributed at random: $[0, 0.6, 0.9, 0, 0.15]$.
- The **dropout rate** is the fraction of the features that are zeroed out and is usually set between 0.2 and 0.5, with input units having a rate closer to 0.2 and hidden units a rate closer to 0.5.
- 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, to balance for the fact that more units are active than at training time. This is called the **weight scaling inference rule**.
- If your dropout rate is 0.5, then you can either divide the weights by 2 at the end of the training, or multiply the output of the units by 2 during training.



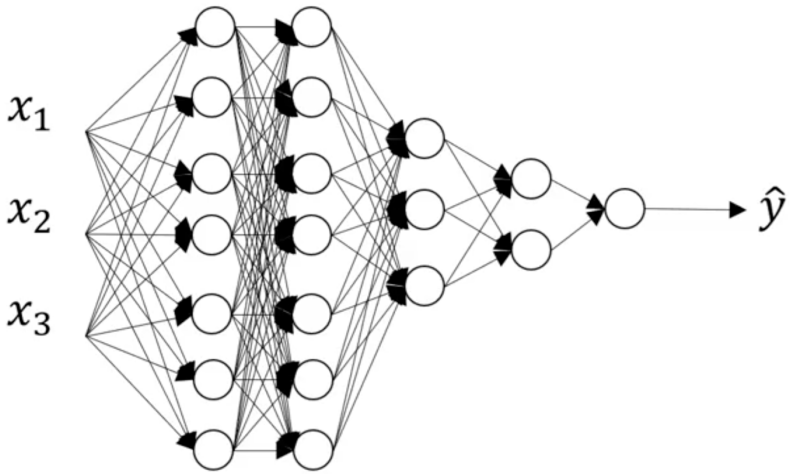
Dropout

- Dropout can also be combined with other forms of regularization to yield further improvement.
- One downside: to offset the effect of reducing the effective capacity of the model, the model size must increase. The validation set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations (epochs) of the training algorithm.
- For large data sets, the computational cost of dropout might outweigh the benefits of regularization

Dropout



Dropout

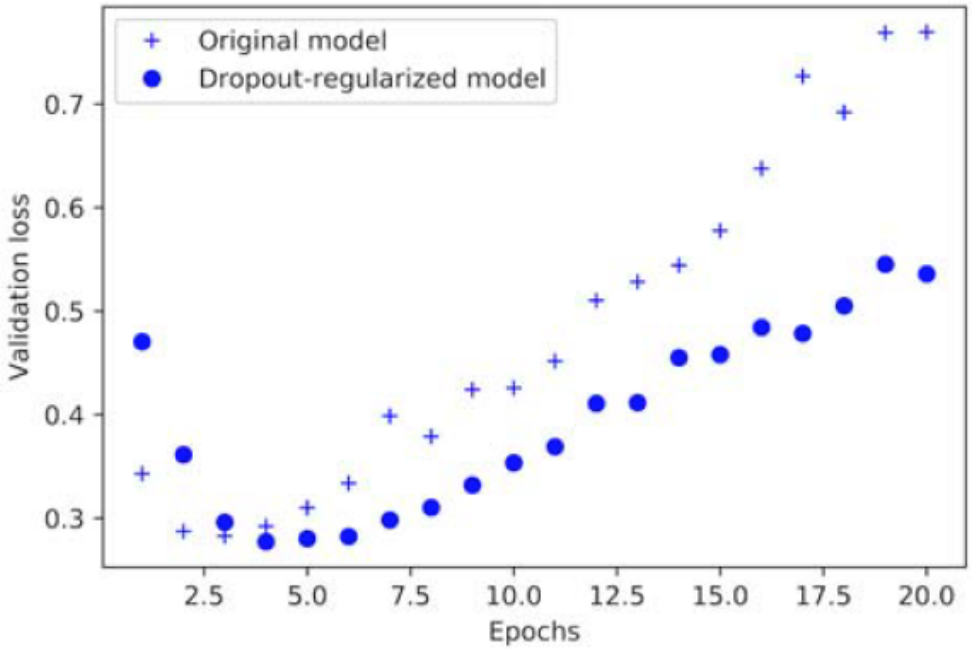


Dropout in Keras

- In Keras, you can introduce dropout in a network via the `Dropout` layer, which is applied to the layer **before** it:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

Dropout in Keras



A Note on Batch Sizes

- Batch, or mini-batch, sizes are generally driven by the following factors:
  - Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
  - Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a mini-batch.
  - If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
  - Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
  - Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

Universal Deep Learning Workflow

1. Define the problem and assemble a data set
   - What will your input data be?
     - What are you trying to predict?
     - What type of problem or task are you facing?
2. Choose a measure of success
   - You need to define what you mean by 'success': accuracy? precision and recall?
     - Your metric of success will guide the choice of loss function and it should directly align with your higher-level goals
     - Note that you can also come up with your own metric by which to measure success
3. Decide on an evaluation protocol
   - Establish how you will measure current progress
     - One of the 3 protocols we've discussed (hold-old validation, K-fold validation and iterated K-fold validation) will work well

Universal Deep Learning Workflow

4. Prepare your data
   - You must format your data in a way that can be fed into a machine-learning model
     - Data should be formatted as tensors
     - The values contained in these tensors should be scaled to small values: usually in the $[0, 1]$ or $[-1, 1]$ range
     - If different features take values in different ranges (heterogeneous data), then the data should be normalized
     - You may need to do some feature engineering
5. Develop a model that does better than baseline
   - The goal is to develop a model that is capable of beating a baseline e.g. randomly assigning a class in a classification task
     - If you can't beat a random baseline after trying multiple reasonable architectures, it may be that the answer to the question you're asking isn't present in the input data
     - By building a model you are assuming (hypothesizing) 2 things:
       - Your outputs can be predicted given your inputs
       - The available data is sufficiently informative to learn the relationship between inputs and outputs
     - You will need to make 3 key choices:
       - Last-layer activation (constrains the network's output)
       - Loss function (should match the type of problem you're trying to solve)
       - Optimization configuration (Which optimizer will you use? What will the learning rate be?)

Universal Deep Learning Workflow

6. Develop a model that overfits
   - In order to figure out how big your model needs to be, you should develop a models that overfits and then scale down
     - To do this:
       - Add layers
       - Make the layers bigger
       - Train for more epochs
     - Always monitor the training and validation loss. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.
7. Regularize your model and tune your hyperparameters
   - This step will take the most time and be an iterative process: modify the model, train it, evaluate, modify again, ...
     - Things you should try:
       - Add dropout
       - Try difference architectures: add or remove layers
       - Add L^1 and/or L^2 regularization
       - Try different hyperparameters (number of hidden units per layer, learning rate of optimizer)
       - Add new features or remove features that don't seem to be informative