

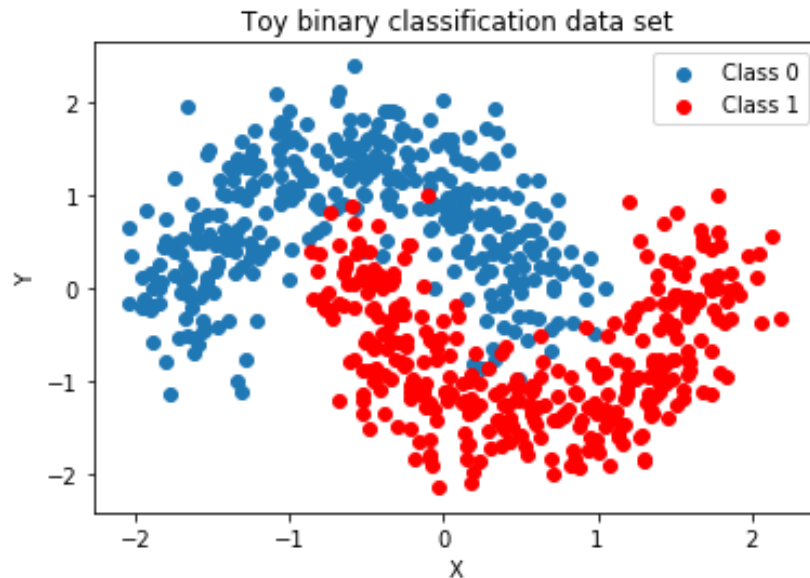
Introduction to Machine Learning

Neural Networks

July 8rd, 2019

**Instructors: Melanie Fernandez Pradier (Harvard), Weiwei Pan (Harvard),
Javier Zazo Ruiz (Harvard)**

Can we build a model that is arbitrarily flexible AND scalable?



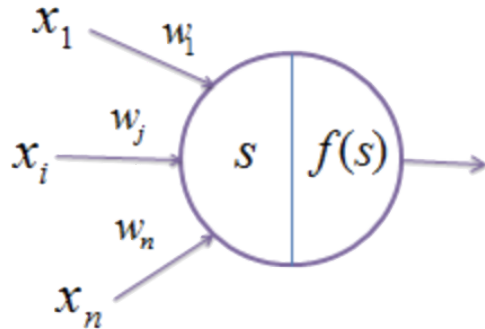
REVISED GOAL: Find models that can capture *arbitrarily complex* trends or decision boundaries **and** are fast to train as well as efficient for computing predictions.

Neural Networks

What is a Neural Network?

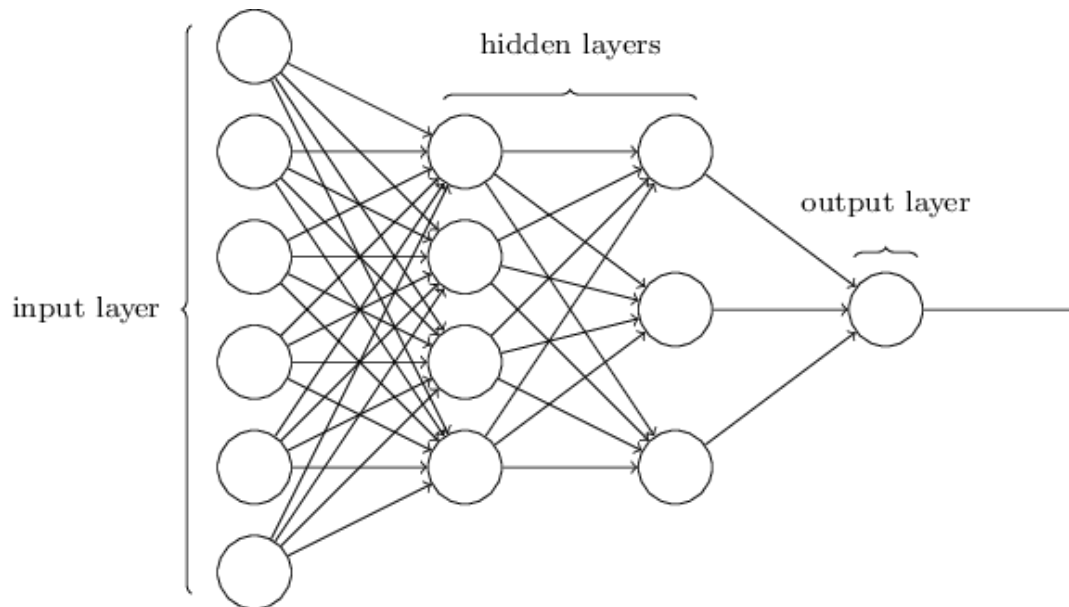
Goal: build a good approximation \hat{g} of a complex function g by composing simple functions.

For example, let the following picture represents $f\left(\sum_i w_i x_i\right)$, where f is a non-linear transform:



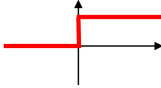
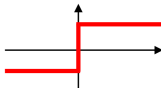
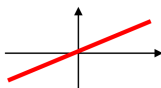

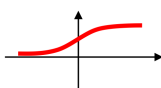
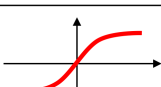
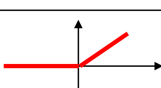

Neural Networks as Function Approximators

Then we can define the approximation \hat{g} with a graphical schema representing a complex series of compositions and sums of the form, $f\left(\sum_i w_i x_i\right)$



This is a **neural network**. We denote the weights of the neural network collectively by \mathbf{W} . The non-linear function f is called the **activation function**.

Common Choices for the Activation Function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Using Neural Networks for Regression

Neural Networks Regression

Data: features x_{train} , real-valued labels y_{train}

Probabilistic Model: $y_{\text{train}} = g_{\mathbf{W}}(x_{\text{train}}) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$, where $g_{\mathbf{W}}$ is a neural network with parameters \mathbf{W} .

Training Objective: find \mathbf{W} to maximize the likelihood of our data. This is equivalent to minimizing the Mean Square Error,

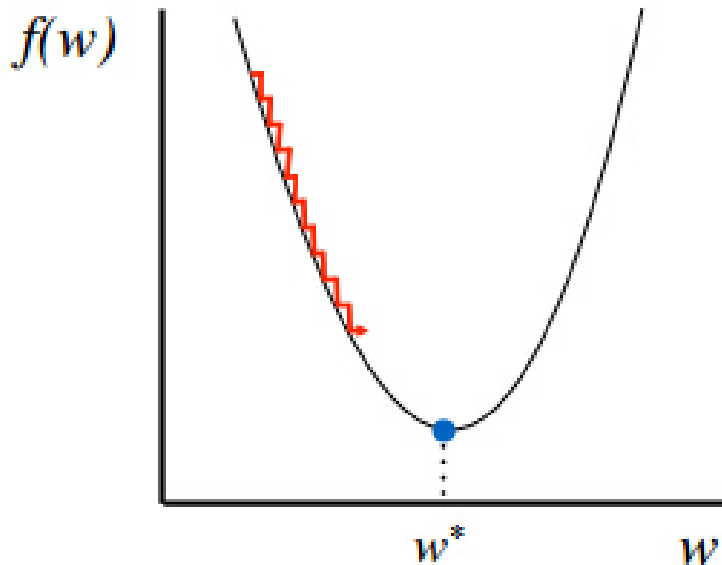
$$\max_{\mathbf{W}} \text{MSE}(\mathbf{W}) = \frac{1}{N} \sum_{n=1}^N (y_n - g_{\mathbf{W}}(x_n))^2$$

Optimizing the Training Objective: For linear regression (when $g_{\mathbf{W}}$ is a linear function), we computed the gradient of the MSE with respect to the model parameters \mathbf{W} , set it equal to zero and solved for the optimal \mathbf{W} analytically.

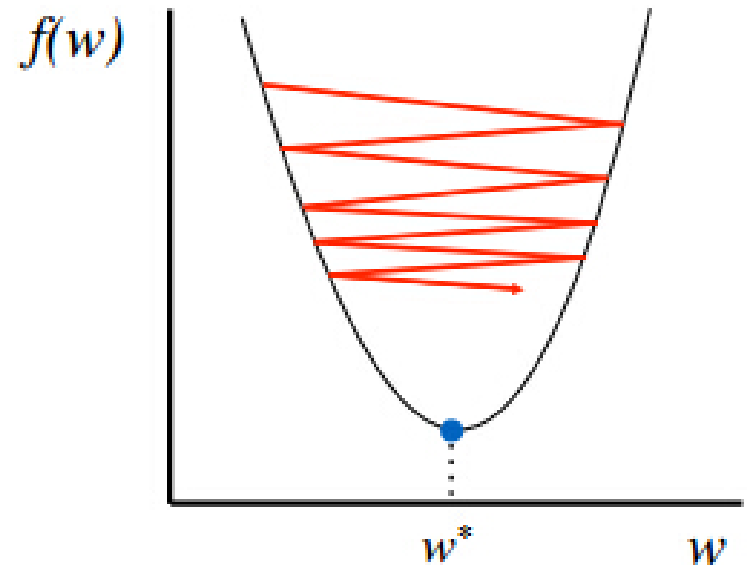
Can we do the same when $g_{\mathbf{W}}$ is a neural network?

Training Your NN → Gradient descent

- Optimization Choices Matter



Too small: converge
very slowly



Too big: overshoot and
even diverge

Gradient Descent: the Algorithm

1. start at random place: $W_0 \leftarrow \text{random}$
2. until (stopping condition satisfied):
 - a. compute gradient: $\text{gradient} = \nabla \text{loss_function}(W_t)$
 - b. take a step in the negative gradient direction: $W_{t+1} \leftarrow W_t - \text{eta} * \text{gradient}$

Here *eta* is called the **learning rate**.

Drawbacks of Gradient Descent

- Consider minimizing an average of functions (over examples):

$$\min_x \frac{1}{N} \sum_{i=1}^N f_W(x_i)$$

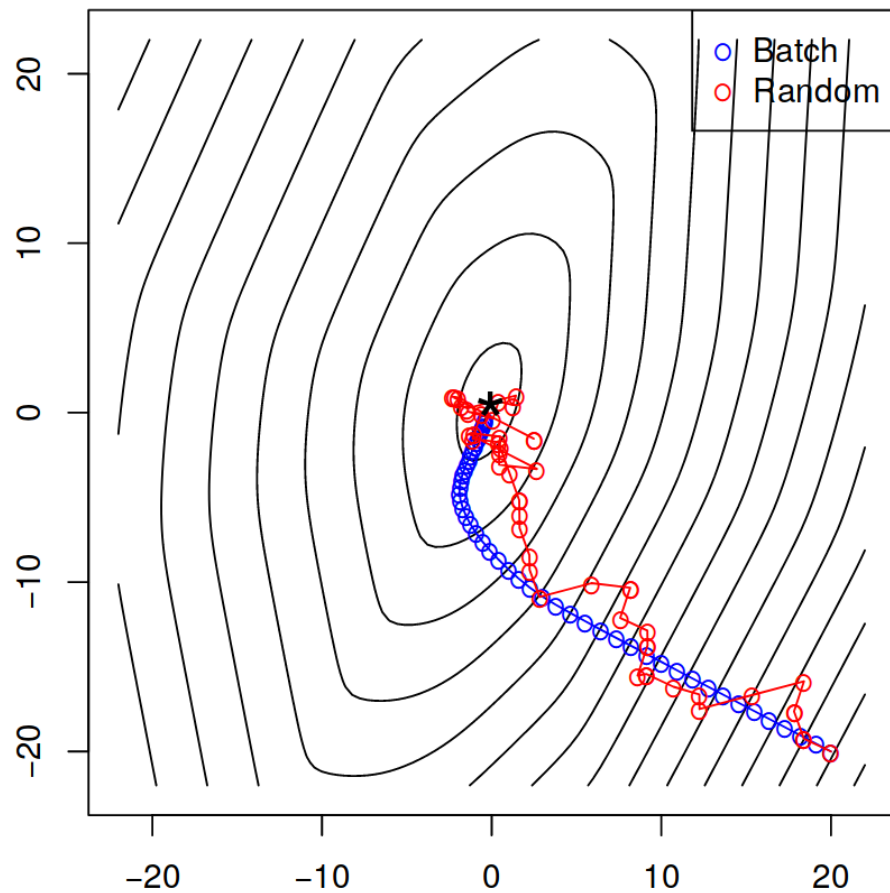
- The gradient update looks:

$$W_{t+1} \leftarrow W_t - \eta * \sum_i \frac{1}{N} \nabla_W f_W(x_i)$$

- What happens if you have a 10k examples? 100k examples? a million examples?
- Is gradient descent still feasible?
- How can we optimize over such database?

Stochastic Gradient Descent

- We could try to use less examples to approximate the gradient...
- With a single example:
$$W_{t+1} \leftarrow W_t - \eta * \nabla_W f_{W_t}(x_i), \quad \forall t = 1, 2, \dots$$
- Every time we make an update on W_t , we take a new example randomly.
- How do you think this affects the optimization procedure?



Mini-batch gradient descent

- Nobody calls it mini-batch gradient descent: SGD
- Instead of using a single example --> use a mini-batch!

$$W_{t+1} \leftarrow W_t - \eta * \frac{1}{M} \sum_{i=1}^M \nabla_W f_{W_t}(x_i), \quad \forall t = 1, 2, \dots$$

- This will reduce the variance of the updates.
- Mini-batch size is another tuning parameter.
- **Epoch:** A whole loop over all data samples.

Implementing Neural Networks in **python**

keras: a Python Library for Neural Networks

keras is a python library that provides intuitive api's for build neural networks quickly.

```
#keras model for feedforward neural networks  
from keras.models import Sequential
```

```
#keras model for layers in feedforward networks  
from keras.layers import Dense
```

```
#keras model for optimizing training objectives  
from keras import optimizers
```


Building a Neural Network for Regression in keras

keras is a python library that provides intuitive api's for build neural networks quickly.

```
#instantiate a feedforward model  
model = Sequential()
```

```
#add layers sequentially
```

```
#input layer: 2 input dimensions  
model.add(Dense(2, input_dim=2, activation='relu',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

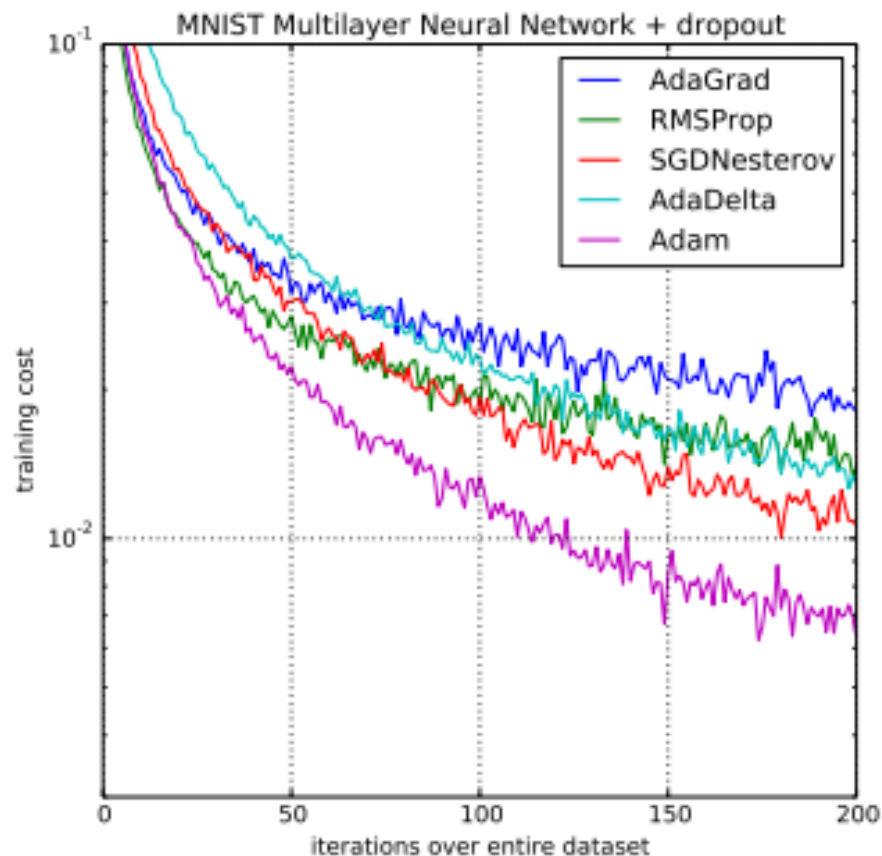
```
#hidden layer: 2 nodes  
model.add(Dense(2, activation='relu',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

```
#output layer: 1 output dimension  
model.add(Dense(1, activation='relu',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

```
#configure the model: specify training objective and training algorithm  
adam = optimizers.Adam(lr=0.01)  
model.compile(optimizer=adam,  
              loss='mean_squared_error')
```

Choosing an optimizer

- Adjust the learning rate dynamically.
- Average of gradients.
- Accelerated versions: increase convergence speed but also variance



Training a Neural Network in keras

#fit the model and return the mean squared error during training

```
history = model.fit(X_train, Y_train, batch_size=20, shuffle=True, epochs=100,  
                    verbose=0)
```

Monitoring Neural Network Training

Visualize the mean square error over the training, this is called the training *trajectory*.

```
#fit the model and return the mean squared error during training
history = model.fit(X_train, Y_train, batch_size=20, shuffle=True, epochs=100,
                    verbose=0)

# Plot the loss function and the evaluation metric over the course of training
fig, ax = plt.subplots(1, 1, figsize=(10, 5))

ax.plot(np.array(history.history['mean_squared_error']), color='blue', label=
        'training accuracy')

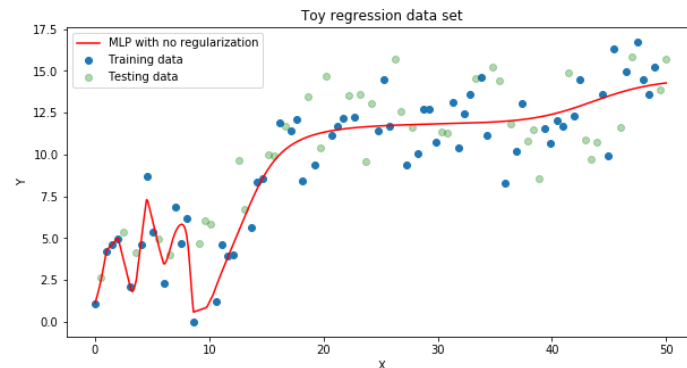
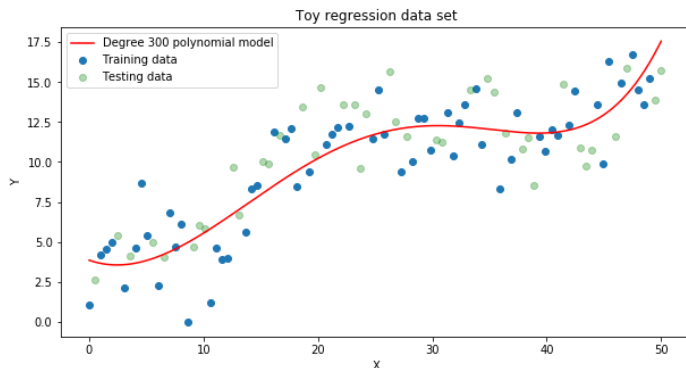
plt.show()
```

The Bias Variance Trade-off: for Neural Networks

Generalization Error and Bias/Variance

Complex models have **low bias** -- they can model a wide range of functions, given enough samples.

But complex models like neural networks can use their 'extra' capacity to explain non-meaningful features of the training data that are unlikely to appear in the test data (i.e. noise). These models have **high variance** -- they are very sensitive to small changes in the data distribution, leading to drastic performance decrease from train to test settings.



Regularization

A way to prevent overfitting is to reduce the capacity of the model, thereby limiting the kinds of functions they can model. This **increases bias, but reduces variance**:

1. ℓ_1, ℓ_2 **weight regularization** - adding a term to the loss function that penalizes the ℓ_1 -norm (sum of absolute values) or the ℓ_2 -norm (sum of squares) of the weights. This prevents the network from learning extremely squiggly functions.

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                  kernel_regularizer=regularizers.l2(0.01),
                  activity_regularizer=regularizers.l1(0.01)))
```

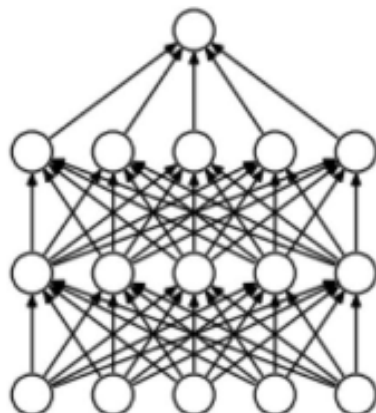
2. **Dropout** - randomly zeroing out weights during training. This prevents the hidden nodes from "over specializing" or "memorizing" certain data points.

```
from keras.layers import Dropout,
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
```

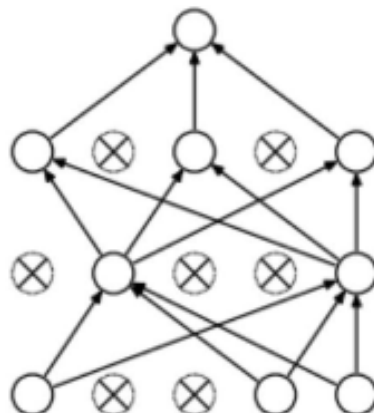
Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov

NITISH@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
KRIZ@CS.TORONTO.EDU
ILYA@CS.TORONTO.EDU
RSALAKHU@CS.TORONTO.EDU



(a) Standard Neural Net



(b) After applying dropout.

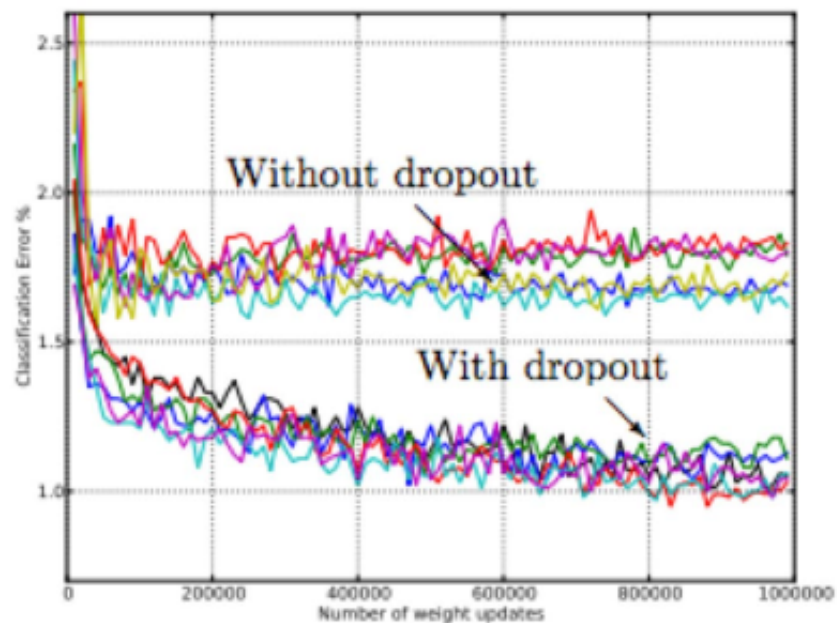


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Exercise: Compare Neural Network Regression to Polynomial Regression

Compare a neural network regression to a polynomial regression. Which model do you think is "better" and for what kind of tasks?

Using Neural Networks for Classification

Data: features x_{train} , real-valued labels y_{train}

Probabilistic Model: $y_{\text{train}} \sim \text{Bernoulli}(\sigma(g_{\mathbf{W}}(x_{\text{train}})))$, where $g_{\mathbf{W}}$ is a neural network with parameters \mathbf{W} , and $\sigma(z) = \frac{1}{1+e^{-z}}$.

Training Objective: find \mathbf{W} to maximize the likelihood of our data. This is equivalent to minimizing the *binary cross entropy* or *log loss*,

$$\max_{\mathbf{W}} \text{CrossEnt}(\mathbf{W}) = \sum_{n=1}^N y_n \log(g_{\mathbf{W}}(x_n)) + (1 - y_n) \log(1 - g_{\mathbf{W}}(x_n))$$

Optimizing the Training Objective: Since this objective is not convex and finding the zero's of the gradient is intractable, we will use gradient descent to find a "optimal" set of parameters \mathbf{W} .

Building a Neural Network for Classification in keras

#instantiate a feedforward model

```
model = Sequential()
```

#add layers sequentially

#input layer: 2 input dimensions

```
model.add(Dense(2, input_dim=2, activation='relu',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

#hidden layer: 2 nodes

```
model.add(Dense(2, activation='relu',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

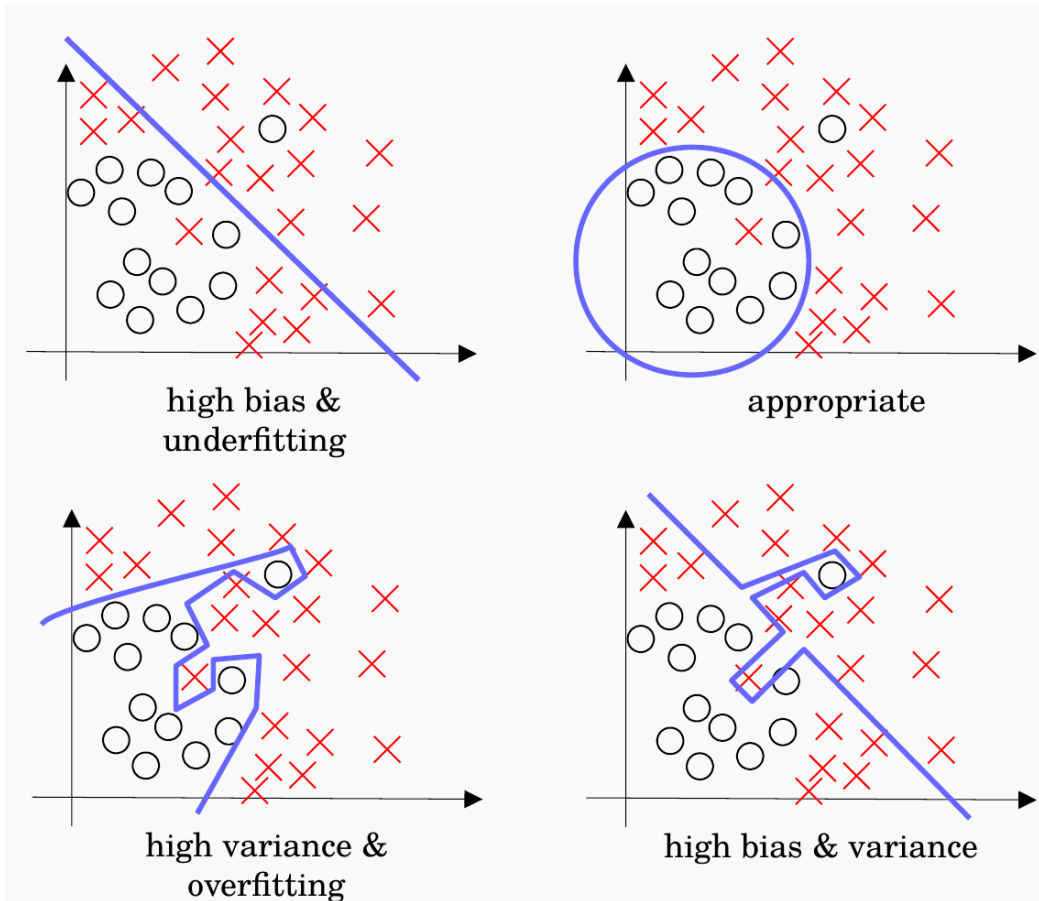
#output layer: 1 output dimension

```
model.add(Dense(1, activation='sigmoid',  
                kernel_initializer='random_uniform',  
                bias_initializer='zeros'))
```

#configure the model: specify training objective and training algorithm

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)  
model.compile(optimizer=sgd,  
              loss='binary_crossentropy')
```

Reminder of bias & variance trade-off in classification



Exercise: Compare Neural Network Classification to Other Classifiers

Compare a neural network classifier to other classifiers. For what task is a neural network the most appropriate model?

KEY IDEAS recap:

- NNs are universal approximators: can approximate ANY function with enough hidden units
- How to we train NN? Gradient descent + chain rule = Back-propagation.
- Wide variety of optimizers --> optimization alternatives.
- Deep NNs can overfit, they WILL overfit!
- Lot's of parameters to tune.
- **Extra note:** Linear/Logistic Regression ARE neural networks with no hidden layers.

How to avoid overfitting?

- more training data
- L2/L1 penalties on weights
- data augmentation
- dropout
- early stopping

Optimizing quality/speed

- learning rate
- batch size

Orthogonalization

Human
level error



Avoidable
bias

Training
error

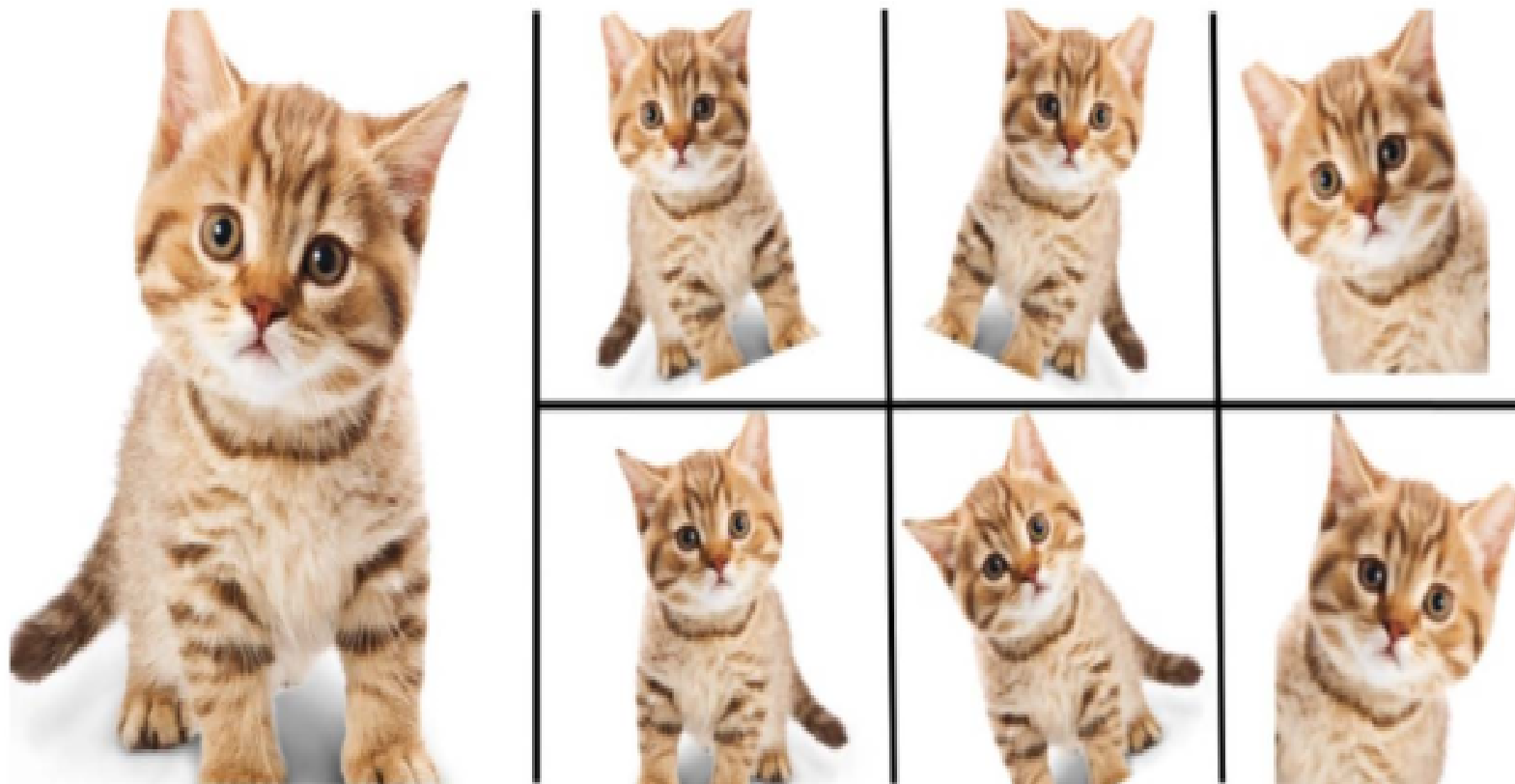


Avoidable
variance

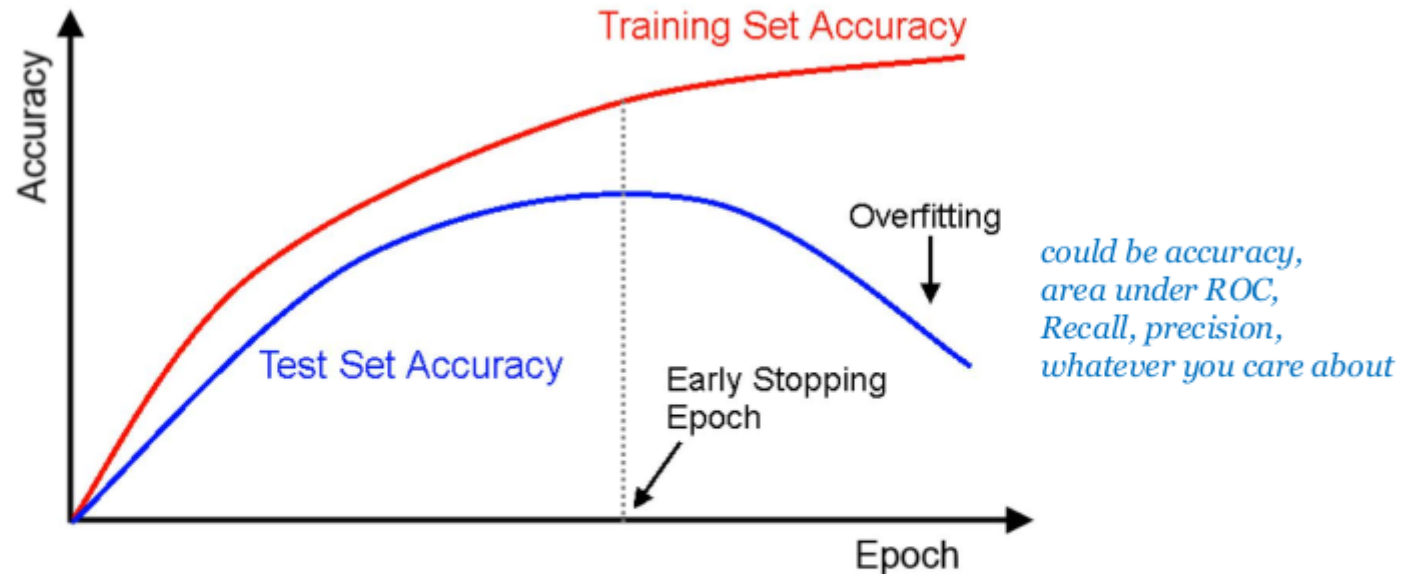
Val error

- Train a bigger model
 - Train longer/better optimization alg.
 - NN architecture/hyperparameter search.
-
- Get more data.
 - Use regularization (L2, dropout, data aug., etc.)
 - NN architecture/hyperparameter search.

Data augmentation



Early stopping



Big idea: stop training after your heldout set stops improving

- Avoid overfitting
- Save time / compute resources

Credit: <https://deeplearning4j.org/docs/latest/deeplearning4j-nn-early-stopping>

Summary advantages/disadvantages:

- Advantages:
 - flexible models
 - high performance (state-of-the-art: language, speech, images)
 - open-source software/several resources
- Disadvantages:
 - require lots of data
 - many tuning params
 - computationally expensive
 - optimization not easy: will it converge? is local minimum enough?
 - will it overfit?