BST 261: Data Science II
Lecture 2
MLPs, Feedforward networks in Python with Keras

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020

Multilayer Perceptrons

Perceptron ⇨ MLP

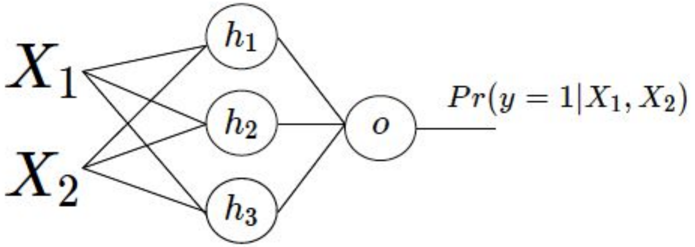
We can turn our perceptron model into a multilayer perceptron

- Instead of just one linear combination, we are going to take several, each with a different set of weights
- Each linear combination will be followed by a nonlinear activation
- Each of these nonlinear features will be fed into the logistic regression classifier (binary classifier)
- All of the weights are learned end-to-end via SGD

MLPs learn a set of nonlinear features directly from data - “feature engineering” is the hallmark of deep learning approaches

Multilayer Perceptrons (MLPs)

Suppose we have the following MLP with 1 hidden layer that has 3 hidden units:



Each neuron in the hidden layer is going to do exactly the same thing as before.

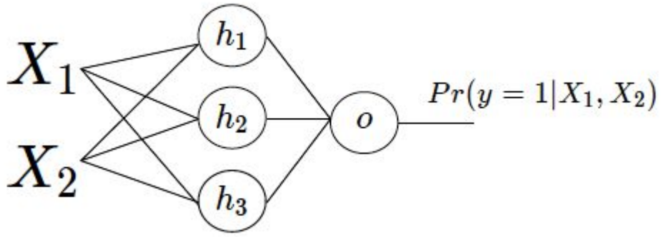
Multilayer Perceptrons (MLPs)

Computations:

$$h_j = \phi(w_{1j} * X_1 + w_{2j} * X_2 + b_j)$$

$$o = b_o + \sum_{j=1}^{3} w_{oj} * h_j$$

$$p = \frac{1}{1+\exp(-o)}$$



*If we use a sigmoid activation function

Output layer weight derivatives

$$\frac{\partial l}{\partial w_{oj}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial w_{oj}}$$

$$= (p - y) * p * (1 - p) * h_j$$

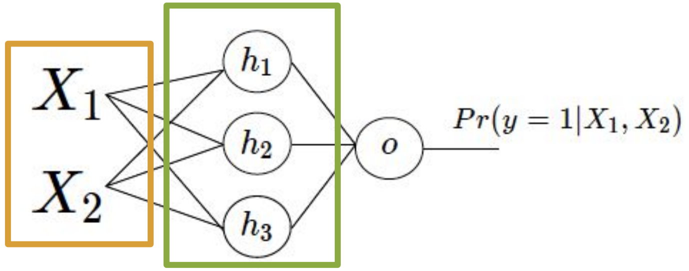
Hidden layer weight derivatives

$$\frac{\partial l}{\partial w_{1j}} = \frac{\partial l}{\partial p} * \frac{\partial p}{\partial o} * \frac{\partial o}{\partial h} * \frac{\partial h}{\partial w_{1j}}$$

$$= (p - y) * p * (1 - p) * h_j * (1 - h_j) * X_1$$

Matrix Notation

Sum notation starts to get unwieldy quickly. We can use matrix notation to represent each calculation in a more concise way.



$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \quad W = \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$Z = W^T X + B \qquad H = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \phi(Z)$$

Notation

As the number of layers grows, the number of matrices grows and we have to add a superscript to denote the layer. We also have to add a superscript to denote which training example we are referencing.

Example notation for 1 weight in 1 hidden layer for 1 training example:

Training example *i*

Layer *l*

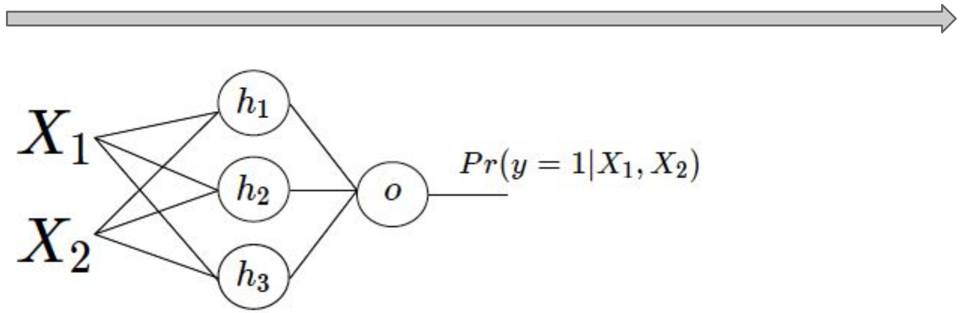
$$w^{[l](i)}_{j,k}$$

Hidden neuron *k*

Input feature *j*

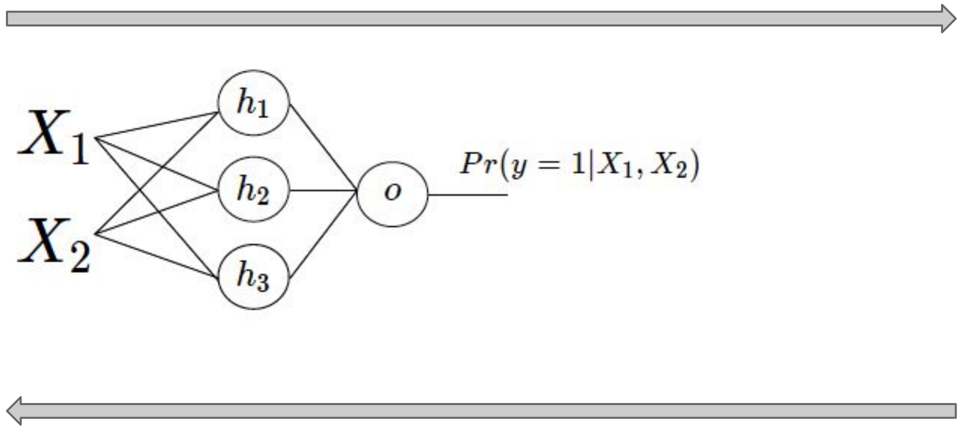
MLP Terminology

Forward pass = computing probability from input



MLP Terminology

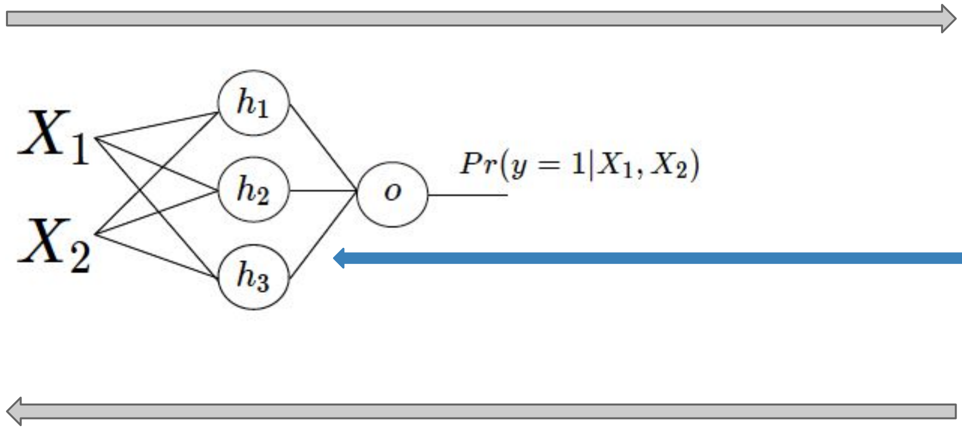
Forward pass = computing probability from input



Backward pass = computing derivatives from the output

MLP Terminology

Forward pass = computing probability from input

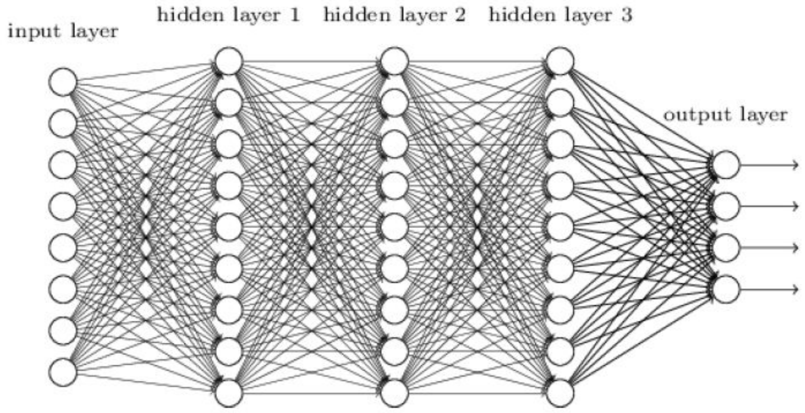


Hidden layers are also called “dense” layers or “fully connected” layers

Backward pass = computing derivatives from the output

MLPs

Increasing the number of layers increases the flexibility of the model - but run the risk of overfitting

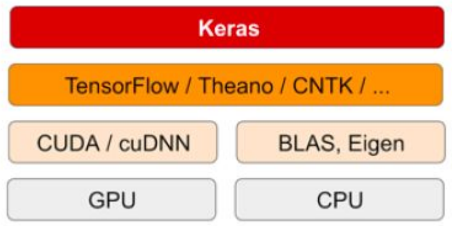


Conclusions

- Backprop, perceptrons, and MLPs are the building blocks of neural nets
- You'll get a chance to demonstrate your mastery in Homework 1
- We will use these concepts for the rest of the semester

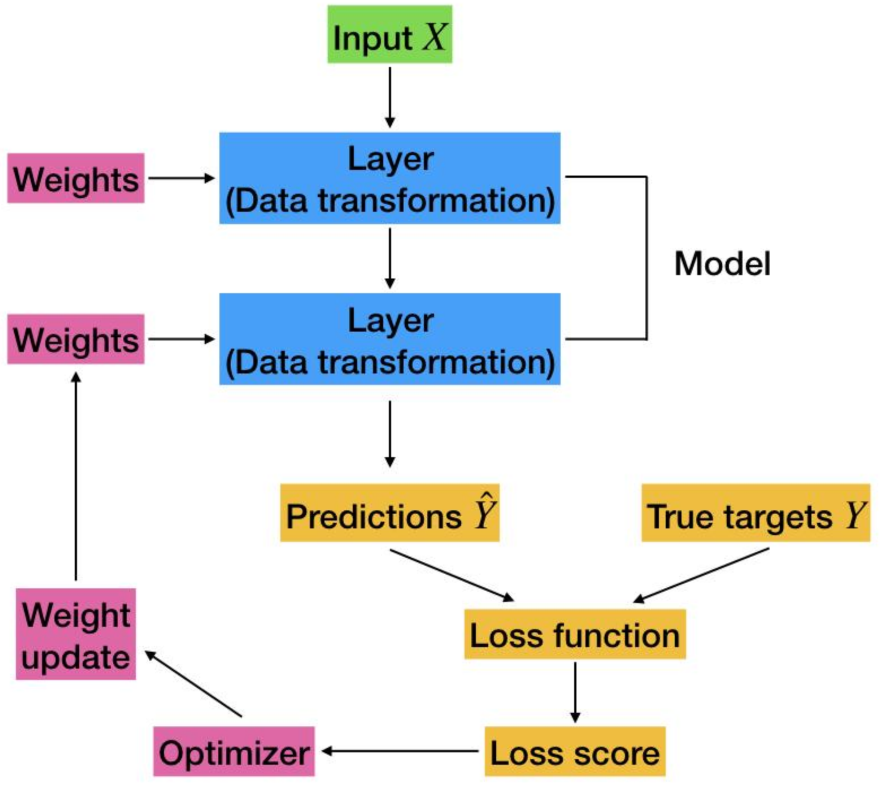
Coding
Neural Nets

Keras and Tensorflow



- Keras is a model-level library that provides high-level building blocks for developing deep learning models
- It doesn't handle low-level operations like matrix and tensor (n-dimensional matrix) multiplication and differentiation
  - It uses TensorFlow or Theano or CNTK (Microsoft Cognitive Toolkit) backends for this
  - We will be using TensorFlow
    - It is the most widely adopted, scalable and production ready
- Keras can run on both CPUs and GPUs
  - When running on CPUs, uses Eigen for tensor operations
  - When running on GPUs, uses the NVIDIA CUDA Deep Neural Network library (cuDNN)

Neural Network Workflow



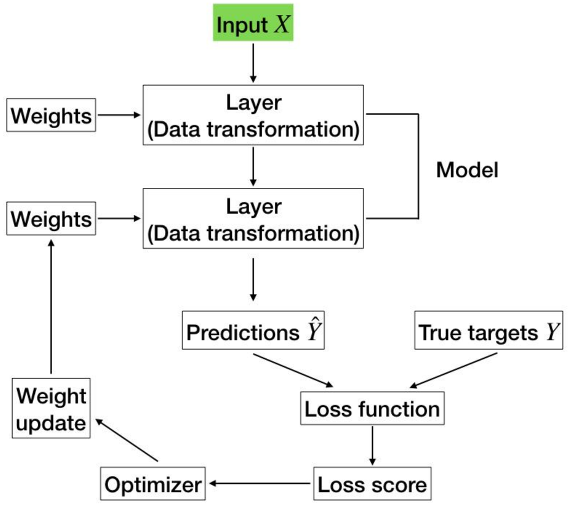
Generic Feedforward Network

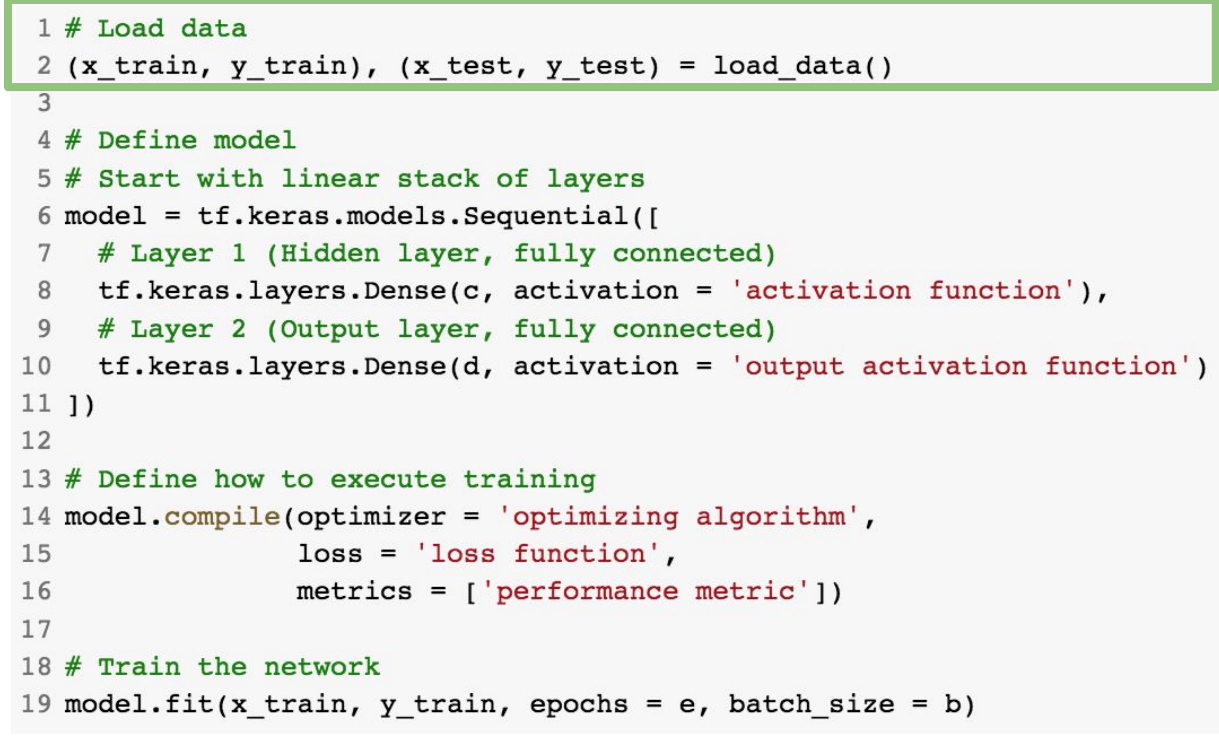
Elements needed:

1. Necessary libraries
2. Dataset split into training and test sets (validation as well if you have enough data)
3. **`models.sequential()`**: defines a linear, or sequential architecture made up of a set of layers that will stack to create the network
4. **`layers.Dense()`**: specifies a fully connected layer
5. **`model.compile(optimizer, loss, metrics)`**: specifies how to execute the training of the network
6. **`model.fit(train_data, train_labels, epochs, batch_size)`**: fits the neural net using the training data, runs for a specified number of iterations (epochs) using batch_size number of training examples at a time

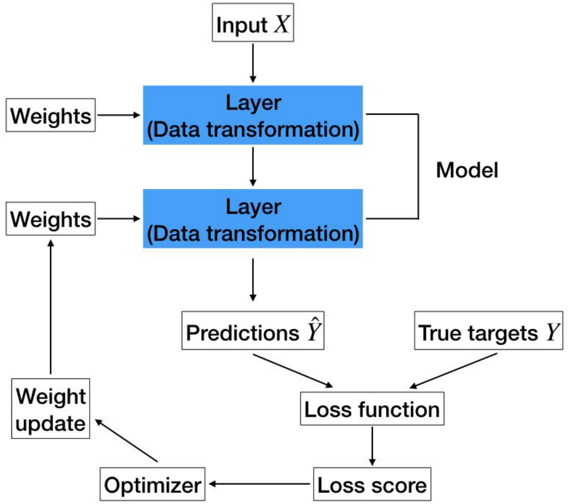
Generic Feedforward Network

```python
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```

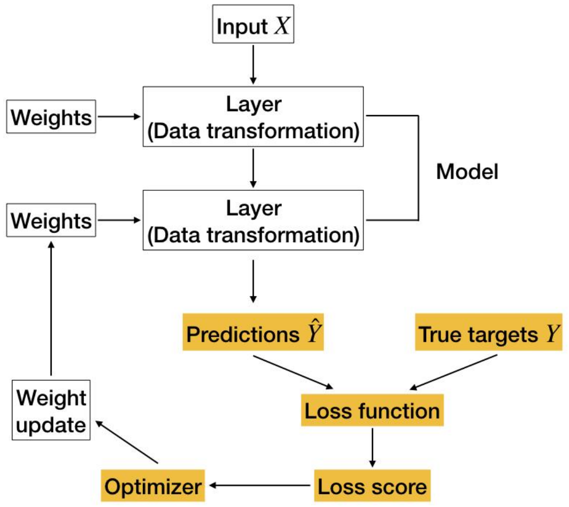




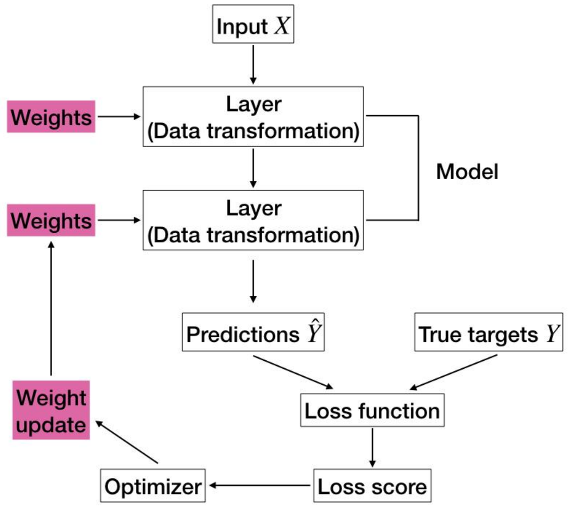
```
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



```python
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



```
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```



```python
# Load data
(x_train, y_train), (x_test, y_test) = load_data()

# Define model
# Start with linear stack of layers
model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer, fully connected)
  tf.keras.layers.Dense(c, activation = 'activation function'),
  # Layer 2 (Output layer, fully connected)
  tf.keras.layers.Dense(d, activation = 'output activation function')
])

# Define how to execute training
model.compile(optimizer = 'optimizing algorithm',
              loss = 'loss function',
              metrics = ['performance metric'])

# Train the network
model.fit(x_train, y_train, epochs = e, batch_size = b)
```

Generic Feedforward Network

train_data: training examples (matrix of feature vectors; $\mathbf{X}_{\mathbf{train}}$)

train_labels: training labels ($\mathbf{y}_{\mathbf{train}}$)

test_data: test examples used to measure performance of network ($\mathbf{X}_{\mathbf{test}}$)

test_labels: test set labels ($\mathbf{y}_{\mathbf{test}}$)

Optimizing algorithms: rmsprop, sgd, adagrad, adam, etc.

Loss function options: mse, mae, categorical_crossentropy, etc.

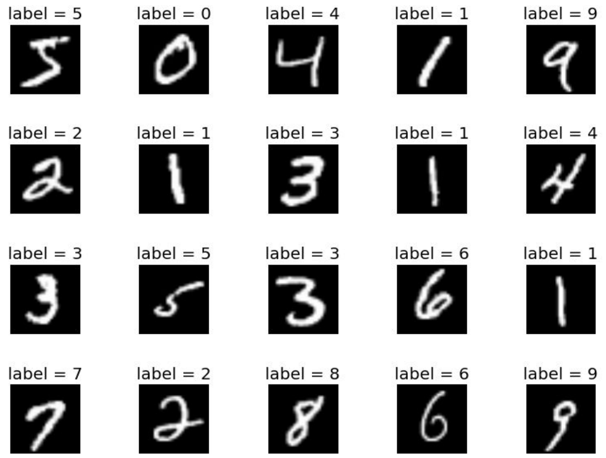
Performance measure options: accuracy, mae, etc.

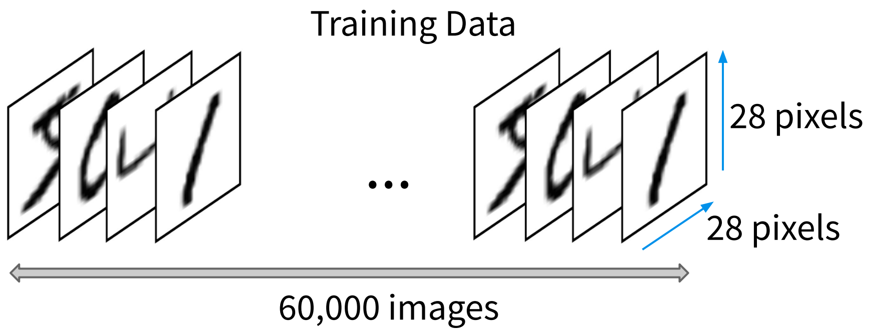
Here:

- ***c*** = the number of hidden units (neurons) in a hidden layer
- ***d*** = the number of units (neurons) in the output layer
- ***e*** = the number of epochs (iterations) over entire training data set
- ***b*** = the batch size (how many training examples to optimize at once)

MNIST Data Example

- The MNIST data set includes handwritten digits with corresponding labels
- Training set: 60,000 images of handwritten digits and corresponding labels
  - Each digit is represented as a 28 x 28 matrix of grayscale values 0 - 255
  - The entire training set is stored in a 3D tensor of shape (60000, 28, 28)
  - The corresponding image values are stored as a 1D tensor of values 0 - 9
- Testing set: 10,000 images with the same set up as the training set





MNIST Data Example

Data wrangling

- We’ll get into RGB images later, but for grayscale images, we need to first transform the matrix of values into a vector of values, and then normalize them to be between 0 and 1. It is not strictly necessary to normalize your inputs, but smaller numbers help speed up training and avoid getting stuck in local minima. This also ensures the gradients don’t “explode” or “vanish”
  - Reshape each image from a 28 x 28 matrix of grayscale values 0 - 255 to a vector of length 28*28 = 784 of values 0 - 1 (divide each by 255)

- We now have 10 classes (categories; the digits 0-9)
  - We need to have multiclass labels that tell the network which digit the example is
  - Reshape each corresponding image label to a vector of length 10 of values 0 or 1
  - Example: the digit 3 would be represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
  - You can think of this as “dummy coding” the labels

Activation and Loss Function Choices

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

Softmax function

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Softmax units are used as outputs when predicting a discrete variable *y* with *k* possible values
- In this setting, which can be seen as a generalization of the Bernoulli distribution, we need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = P(y = i|x)$
- We require that each $\hat{y}_i$ lie in the [0, 1] interval and that the entire vector sums to 1
- We first compute $z = w^T x + b$ as usual
- Here, $z_i = log[\tilde{P}(y = i|x)]$ represents an unnormalized log probability for class *i*
- The softmax function then exponentiates and normalizes *z* to obtain $\hat{\mathbf{y}}$

Softmax function

- In this case we want to maximize

$$log[P(y=i;z)] = log[\text{softmax}(z)_i] = z_i - log\sum_j exp(z_j)$$

- The first term shows that the input always has a direct contribution to the loss function

- Because $log\sum_j exp(z_j) \approx max_j z_j$, the negative log-likelihood loss function always strongly penalizes the most active incorrect prediction

MNIST Data Example

Network Architecture

- Let’s start with 2 layers:
  - Hidden layer will have 512 hidden units and the **relu activation function**

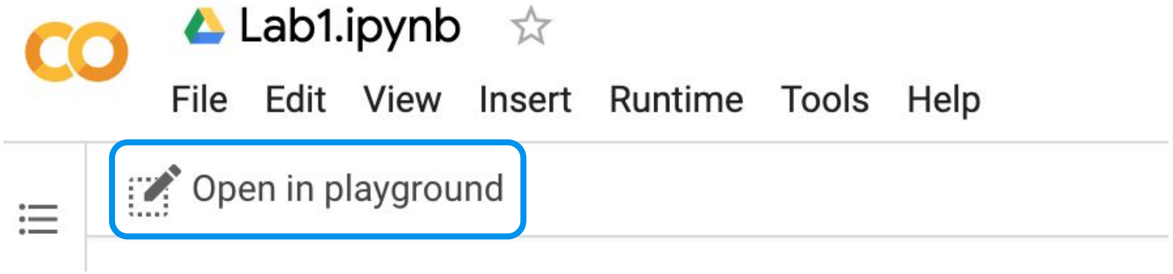
  - Output layer with 10 units (one for each possible digit) and the **softmax activation function** (this produces a vector of length 10, where each element is a probability between 0 and 1 of the image being classified as that digit)
  - Example: [0, 0.3, 0, 0, 0, 0, 0, 0.7, 0, 0] - the highest probability corresponds to a label of 7, so the network would classify this image as a 7

  - **rmsprop optimization algorithm**
  - **categorical_crossentropy loss function**
  - **accuracy performance measure** (the proportion of times the correct class is chosen)

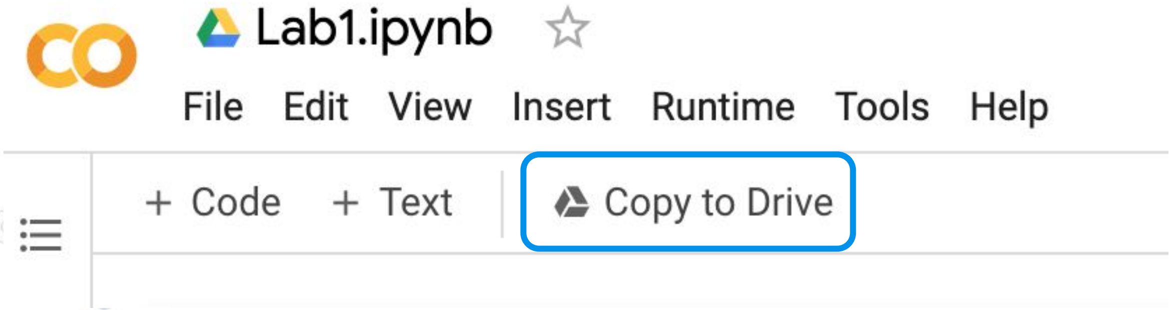
MNIST Data Example

Colab link

Step 1



Step 2



IMDb Data Example

The IMDb data set is a set of movie reviews that have been labeled as either positive or negative, based on the text content of the reviews

- Training set: 25,000 either positive or negative movie reviews that have each been turned into a vector of integers
    - We'll see how to actually do this later in the course
    - Each review can be of any length
    - Only the top 10,000 most frequently occurring words are kept i.e. rare words are discarded
    - Each review includes a label: 0 = negative review and 1 = positive review

- Testing set: 25,000 either positive or negative movie reviews, similar to the training set

IMDb Data Example

Data Wrangling

- Each review is of a varying length and is a list of integers - we need to turn this into a tensor with a common length for each review
- Create a 2D tensor of shape 25,000 x 10,000
  - 25,000 reviews and 10,000 possible words
- Use the **vectorize_sequences** function to turn a movie review list of integers into a vector of length 10,000 with 1s for each word that appears in the review and 0s for words that do not
- The labels are already 0s and 1s, so the only thing we need to do is make them float numbers

Activation and Loss Function Choices

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

IMDb Data Example

Network Architecture

- 3 layers
    - 2 hidden layers and 1 output layer
    - Hidden layers have 16 hidden units each and a **relu activation function**
    - Output layer has 1 unit (the probability a review is positive)
- **Sigmoid activation function**
- **rmsprop optimization algorithm**
- **binary_crossentropy loss function**
- **accuracy performance measure** (proportion of times the correct class is chosen)

IMDb Data Example

Colab link

Regularization

Regularization

- One of the biggest problems with neural networks is overfitting.
- Regularization schemes combat overfitting in a variety of different ways

A perceptron represents the following optimization problem:

$$\text{argmin}_W l(y, f(X)) \quad \text{where} \quad f(X) = \frac{1}{1+\exp(-\phi(XW))}$$

Regularization

One way to regularize is to introduce penalties and change

$$\text{argmin}_W l(y, f(X))$$

to

$$\text{argmin}_W l(y, f(X)) + \lambda R(W)$$

where R(W) is often the L1 or L2 norm of W. These are the well-known ridge and LASSO penalties, and referred to as **weight decay** by the neural net community.

L2 Regularization

We can limit the size of the L2 norm of the weight vector:

$$\text{argmin}_W l(y, f(X)) + \lambda \|W\|_2$$

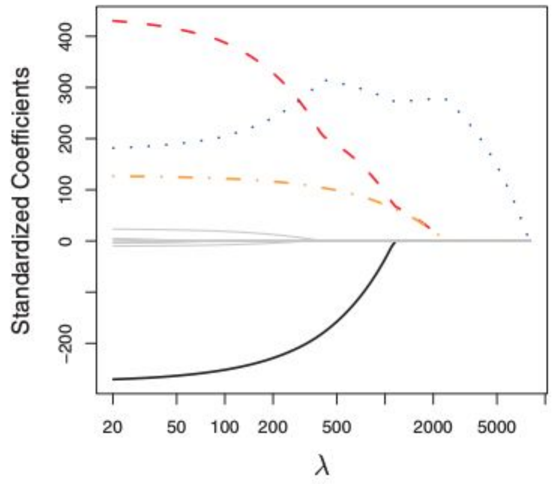
where

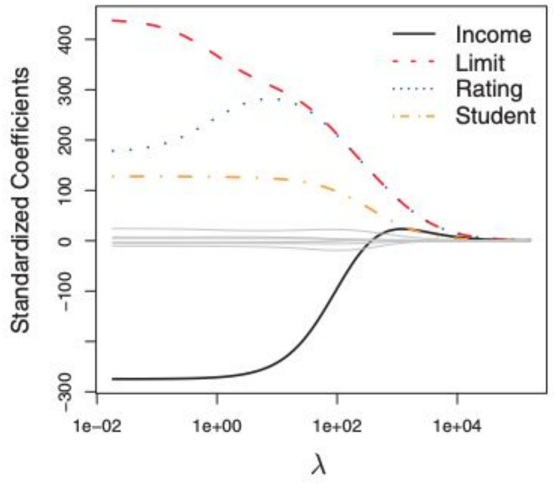
$$\|W\|_2 = \sum_{j=1}^{p} w_j^2$$

We can do the same for the L1 norm. What do the penalties do?

Shrinkage

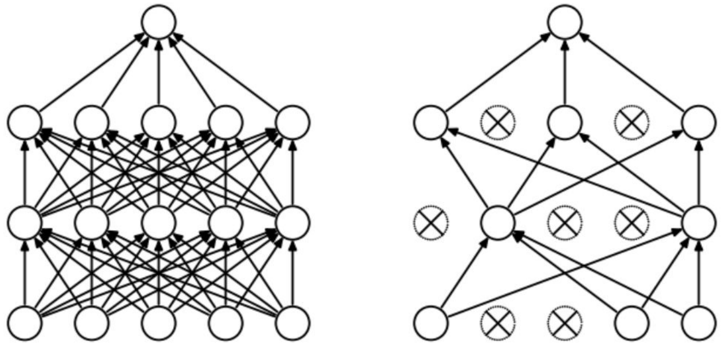
The L1 and L2 penalties shrink the weights to or towards 0.





Stochastic Regularization

- Why is this a good idea?
- Often, we will inject noise into the neural network during training
    - The most popular way to do this is **dropout**
- Given a hidden layer, we are going to set each element of the hidden layer to 0 with probability *p* each SGD update.

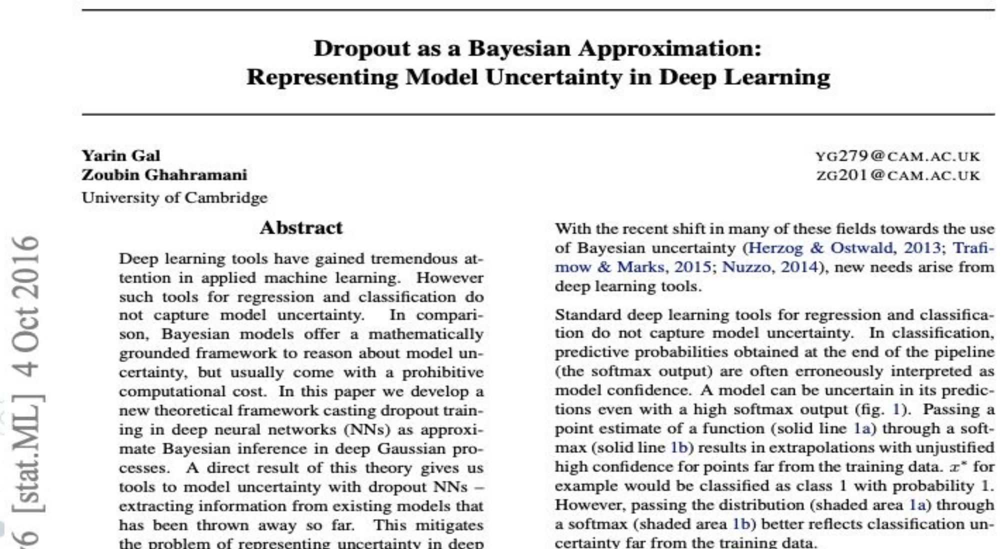


(a) Standard Neural Net (b) After applying dropout.

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Stochastic Regularization

- One way to think of this is the network is trained by bagged versions of the network.
- **Bagging** reduces variance.
- Others have argued this is an approximate Bayesian model



Dropout as a Bayesian Approximation:
Representing Model Uncertainty in Deep Learning

Yarin Gal YG279@CAM.AC.UK
Zoubin Ghahramani ZG201@CAM.AC.UK
University of Cambridge

6 [stat.ML] 4 Oct 2016

Abstract

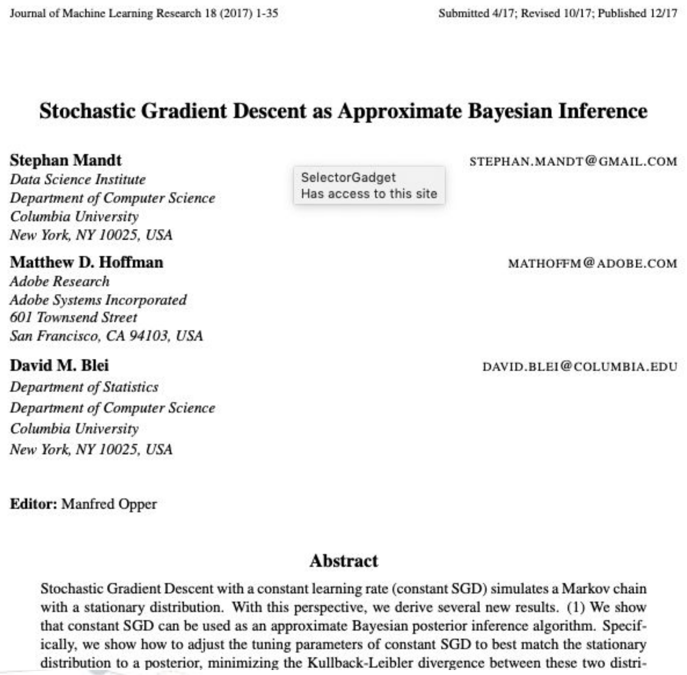
Deep learning tools have gained tremendous attention in applied machine learning. However such tools for regression and classification do not capture model uncertainty. In comparison, Bayesian models offer a mathematically grounded framework to reason about model uncertainty, but usually come with a prohibitive computational cost. In this paper we develop a new theoretical framework casting dropout training in deep neural networks (NNs) as approximate Bayesian inference in deep Gaussian processes. A direct result of this theory gives us tools to model uncertainty with dropout NNs – extracting information from existing models that has been thrown away so far. This mitigates the problem of representing uncertainty in deep

With the recent shift in many of these fields towards the use of Bayesian uncertainty (Herzog & Ostwald, 2013; Trafimow & Marks, 2015; Nuzzo, 2014), new needs arise from deep learning tools.

Standard deep learning tools for regression and classification do not capture model uncertainty. In classification, predictive probabilities obtained at the end of the pipeline (the softmax output) are often erroneously interpreted as model confidence. A model can be uncertain in its predictions even with a high softmax output (fig. 1). Passing a point estimate of a function (solid line 1a) through a softmax (solid line 1b) results in extrapolations with unjustified high confidence for points far from the training data. x^* for example would be classified as class 1 with probability 1. However, passing the distribution (shaded area 1a) through a softmax (shaded area 1b) better reflects classification uncertainty far from the training data.

Stochastic Regularization

- Many have argued that SGD itself provides regularization



Journal of Machine Learning Research 18 (2017) 1-35 Submitted 4/17; Revised 10/17; Published 12/17

Stochastic Gradient Descent as Approximate Bayesian Inference

Stephan Mandt STEPHAN.MANDT@GMAIL.COM
Data Science Institute
Department of Computer Science
Columbia University
New York, NY 10025, USA

SelectorGadget
Has access to this site

Matthew D. Hoffman MATHOFFM@ADOBE.COM
Adobe Research
Adobe Systems Incorporated
601 Townsend Street
San Francisco, CA 94103, USA

David M. Blei DAVID.BLEI@COLUMBIA.EDU
Department of Statistics
Department of Computer Science
Columbia University
New York, NY 10025, USA

Editor: Manfred Opper

Abstract

Stochastic Gradient Descent with a constant learning rate (constant SGD) simulates a Markov chain with a stationary distribution. With this perspective, we derive several new results. (1) We show that constant SGD can be used as an approximate Bayesian posterior inference algorithm. Specifically, we show how to adjust the tuning parameters of constant SGD to best match the stationary distribution to a posterior, minimizing the Kullback-Leibler divergence between these two distri-

Initialization Regularization

- The weights in a neural network are given random values initially.
- There is an entire literature on the best way to do this initialization
  - Normal
  - Truncated Normal
  - Uniform
  - Orthogonal
  - Scaled by number of connections
  - Etc.
- Try to “bias” the model into initial configurations that are easier to train

Initialization Regularization

- A popular way is to do **transfer learning**

Train the model on auxiliary task where lots of data is available

$\Rightarrow$

Use final weight values from previous task as initial values and “fine tune” on primary task

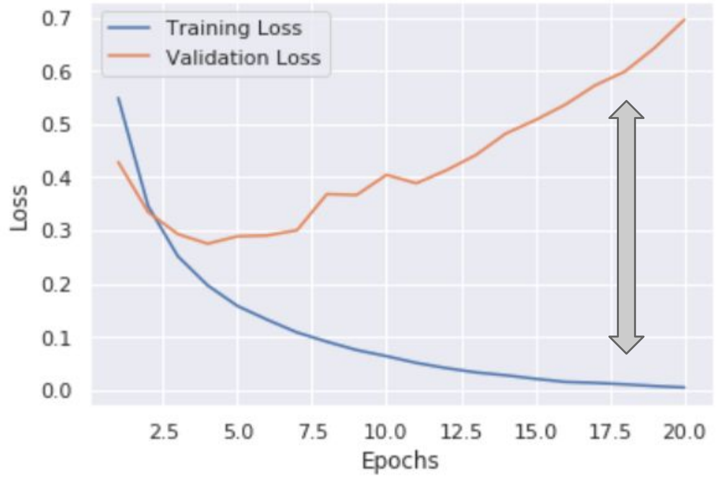
Structural Initialization

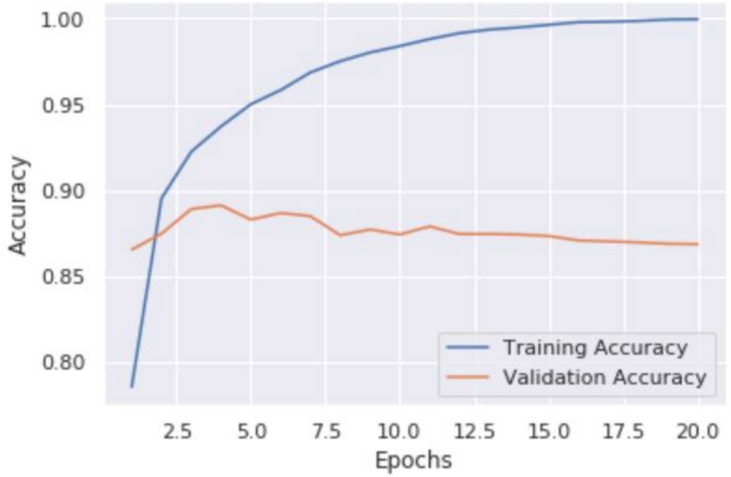
- The key advantage of neural nets is the ability to easily include properties of the data directly into the model through the network's structure

- Convolutional neural networks (CNNs) are a prime example of this

IMDb Example

We saw overfitting in the IMDb example:





How do we make this model better?

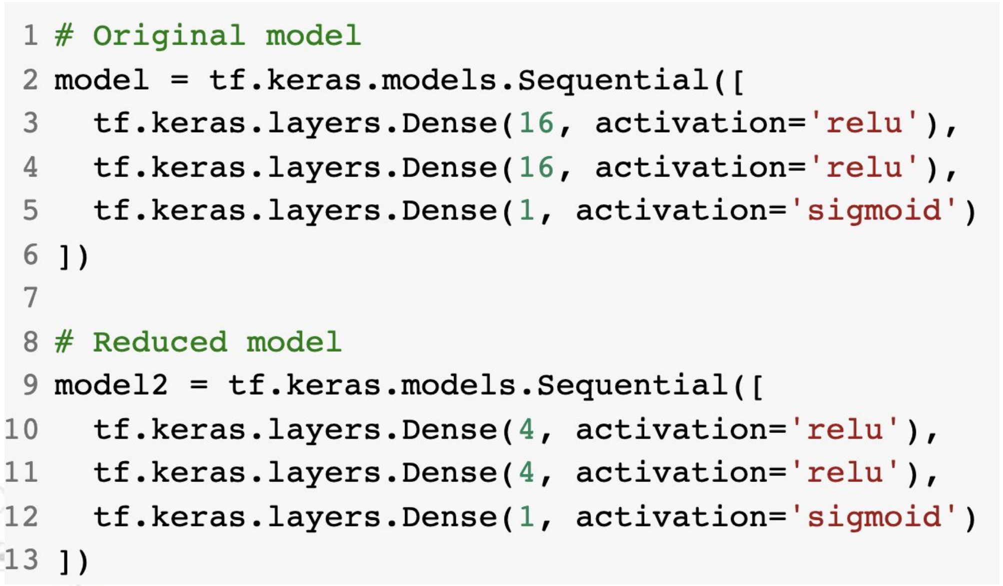
Regularization

1. Reduce network size
2. Weight regularization
3. Dropout

Back to the IMDb colab

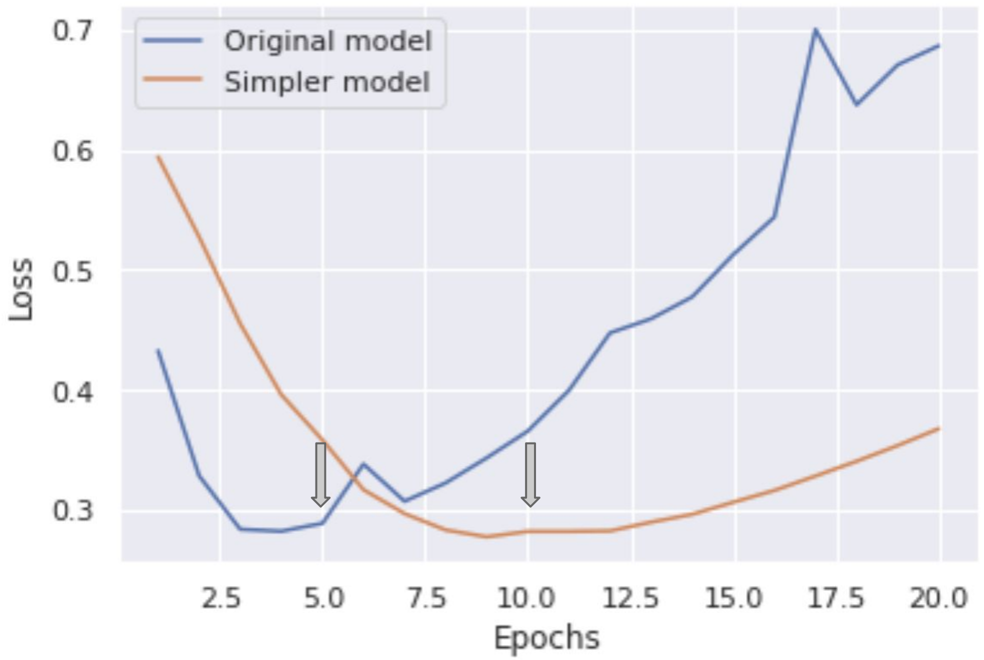
Regularization: reducing network size

When we are battling overfitting, one option is to simplify the model. Let’s compare the performance we get from a simpler model. Here we have simplified the model by reducing the number of hidden units in each hidden layer.



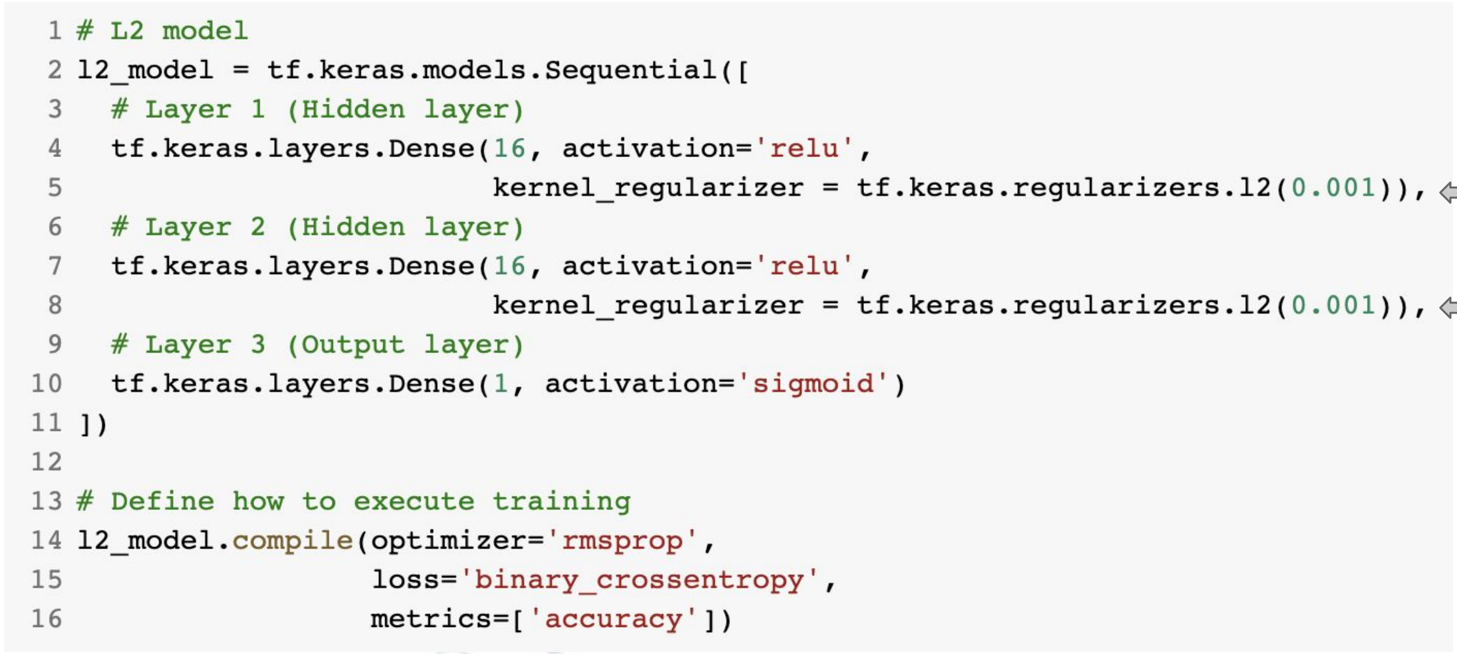
```
# Original model
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

# Reduced model
model2 = tf.keras.models.Sequential([
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(4, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Regularization: reducing network size



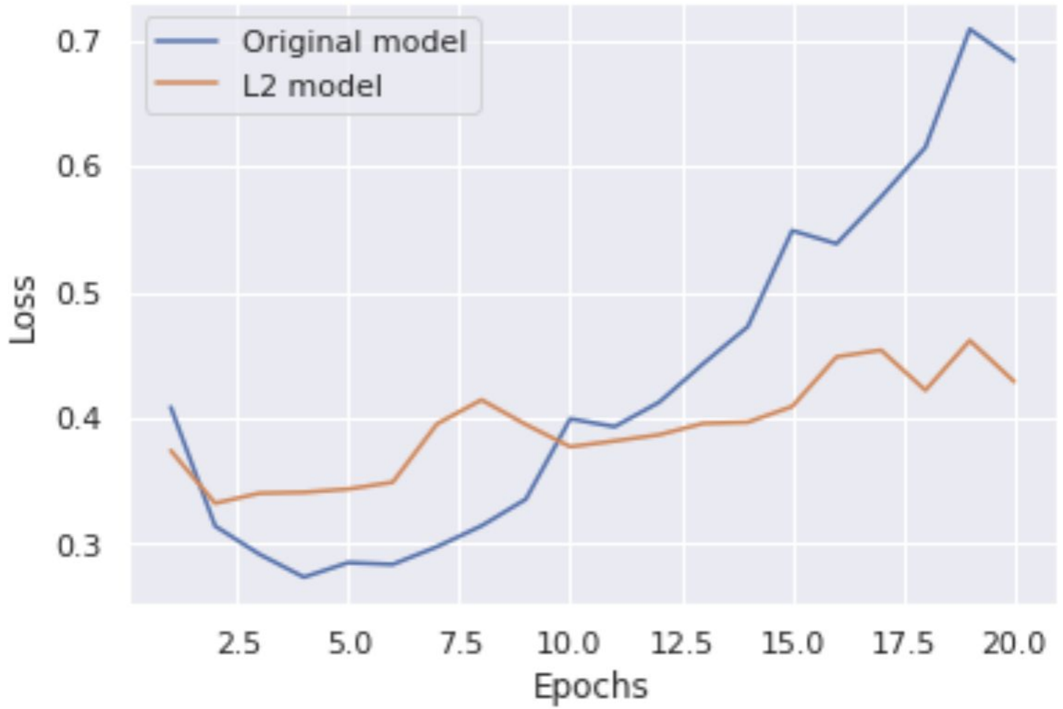
The smaller network performs better than the original model - it starts to overfit at epoch 10 rather than epoch 6. These values are when the validation loss starts to increase.

Regularization: weight regularization



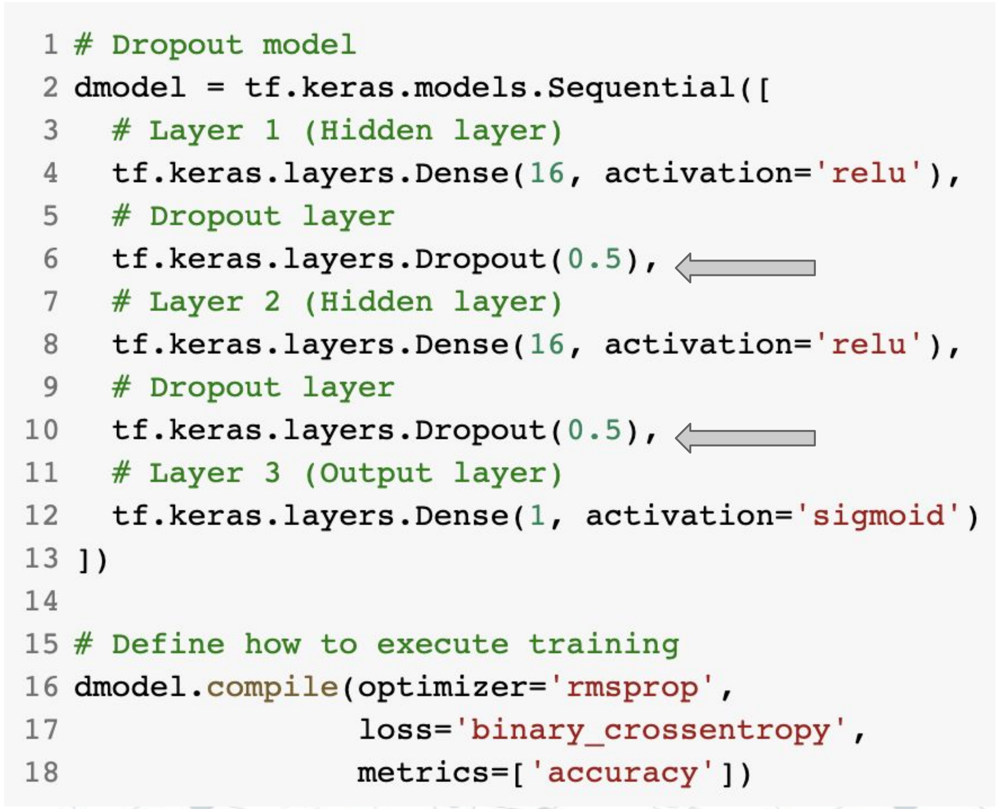
```python
# L2 model
l2_model = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer)
  tf.keras.layers.Dense(16, activation='relu',
                        kernel_regularizer = tf.keras.regularizers.l2(0.001)),
  # Layer 2 (Hidden layer)
  tf.keras.layers.Dense(16, activation='relu',
                        kernel_regularizer = tf.keras.regularizers.l2(0.001)),
  # Layer 3 (Output layer)
  tf.keras.layers.Dense(1, activation='sigmoid')
])

# Define how to execute training
l2_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['accuracy'])
```

Regularization: weight regularization



The L2-regularized model is much more resistant to overfitting - the validation loss starts to increase at a much slower rate

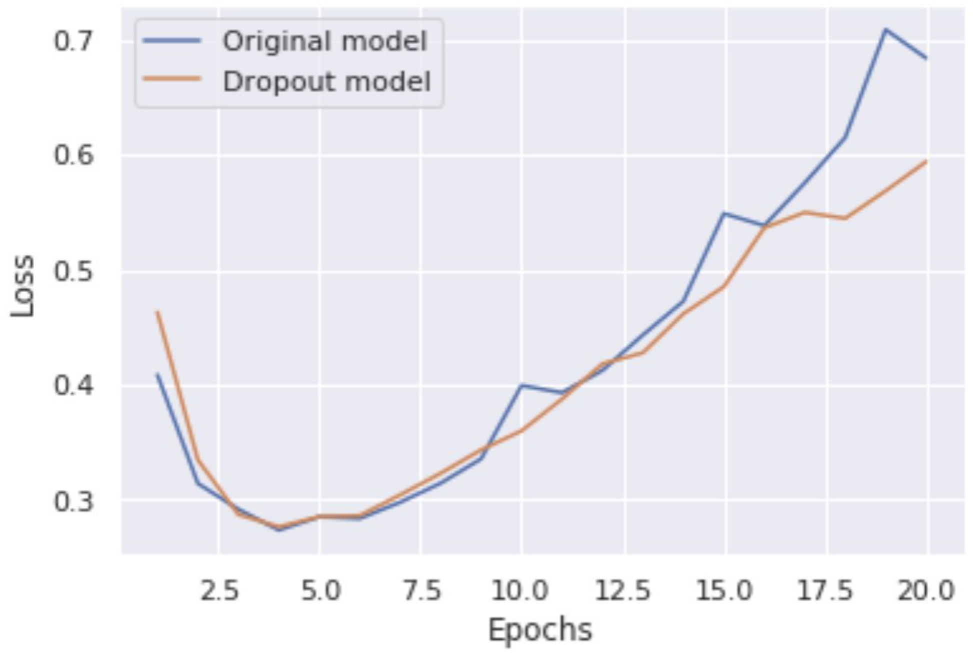
Regularization: adding dropout



```python
# Dropout model
dmodel = tf.keras.models.Sequential([
  # Layer 1 (Hidden layer)
  tf.keras.layers.Dense(16, activation='relu'),
  # Dropout layer
  tf.keras.layers.Dropout(0.5),
  # Layer 2 (Hidden layer)
  tf.keras.layers.Dense(16, activation='relu'),
  # Dropout layer
  tf.keras.layers.Dropout(0.5),
  # Layer 3 (Output layer)
  tf.keras.layers.Dense(1, activation='sigmoid')
])

# Define how to execute training
dmodel.compile(optimizer='rmsprop',
               loss='binary_crossentropy',
               metrics=['accuracy'])
```

The 0.5 indicates a 50% probability of dropping out a unit. Typically, 20% is used in practice but you can try different values and see what performs best.

Regularization: adding dropout



The dropout model is slightly better than the original model but does not control for overfitting as well as the L2 network