Neural Networks in practice

Joaquin Vanschoren, Eindhoven University of Technology

Overview

- Solving basic classification and regression problems
- Handling textual data
- Model selection (and overfitting)

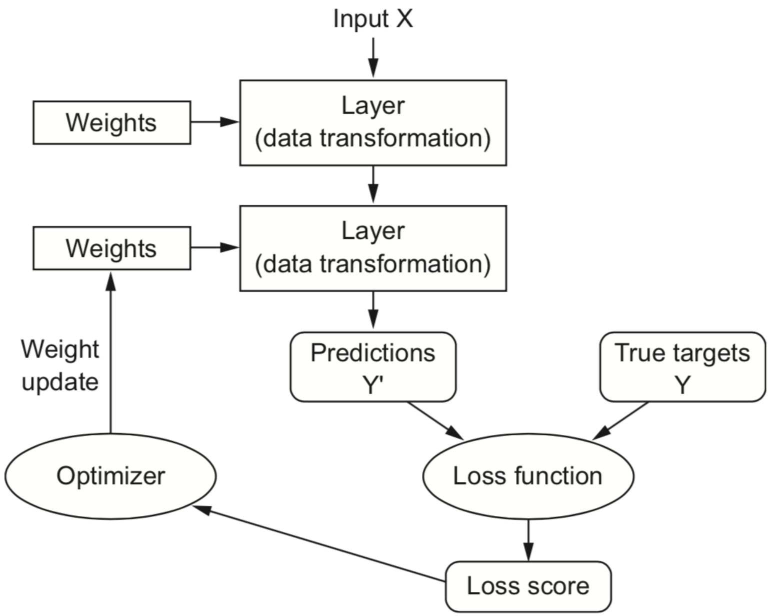
Solving basic problems

- Binary classification (of movie reviews)
- Multiclass classification (of news topics)
- Regression (of house prices)

Examples from *Deep Learning with Python*, by *François Chollet*

Components of Neural Nets (recap)

- *Layers* of nodes: transform an input tensor to an output tensor
  - Each with (a tensor of) weights to be fitted to the training data
  - Many types: dense, convolutional, recurrent,...
- *Loss function*: Measures whether the model fits the training data
- *Optimizer*: How to update the network, e.g. SGD



Binary classification

- Dataset: 50,000 IMDB reviews, labeled positive (1) or negative (0)
    - Included in Keras, with a 50/50 train-test split
- Each row is one review, with only the 10,000 most frequent words retained
- Each word is replaced by a *word index* (word ID)

```
Encoded review:  [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
Original review:  ? this film was just brilliant casting location scenery
story
```

Preprocessing

- We can't input lists of categorical value to a neural net, we need to create tensors
- One-hot-encoding:
    - 10000 features, '1.0' if the word occurs
- Word embeddings (word2vec):
    - Map each word to a dense vector that represents it (it's *embedding*)
    - *Embedding* layer: pre-trained layer that looks up the embedding in a dictionary
    - Converts 2D tensor of word indices (zero-padded) to 3D tensor of embeddings
- Let's do One-Hot-Encoding for now. We'll come back to *Embedding* layers.
- Also vectorize the labels: from 0/1 to float
    - Binary classification works with one output node

```
# Custom implementation of one-hot-encoding
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

```
Encoded review:  [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
One-hot-encoded review:  [0. 1. 1. 0. 1. 1. 1. 1. 1. 1.]
Label:  1.0
```

Building the network

- We can solve this problem using a network of *Dense* layers and the *ReLU* activation function.
- How many layers? How many hidden units for layer?
    - Start with 2 layers of 16 hidden units each
    - We'll optimize this soon
- Output layer: single unit with *sigmoid* activation function
    - Close to 1: positive review, close to 0: negative review

```
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'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[
'accuracy'])
```

Cross-entropy loss

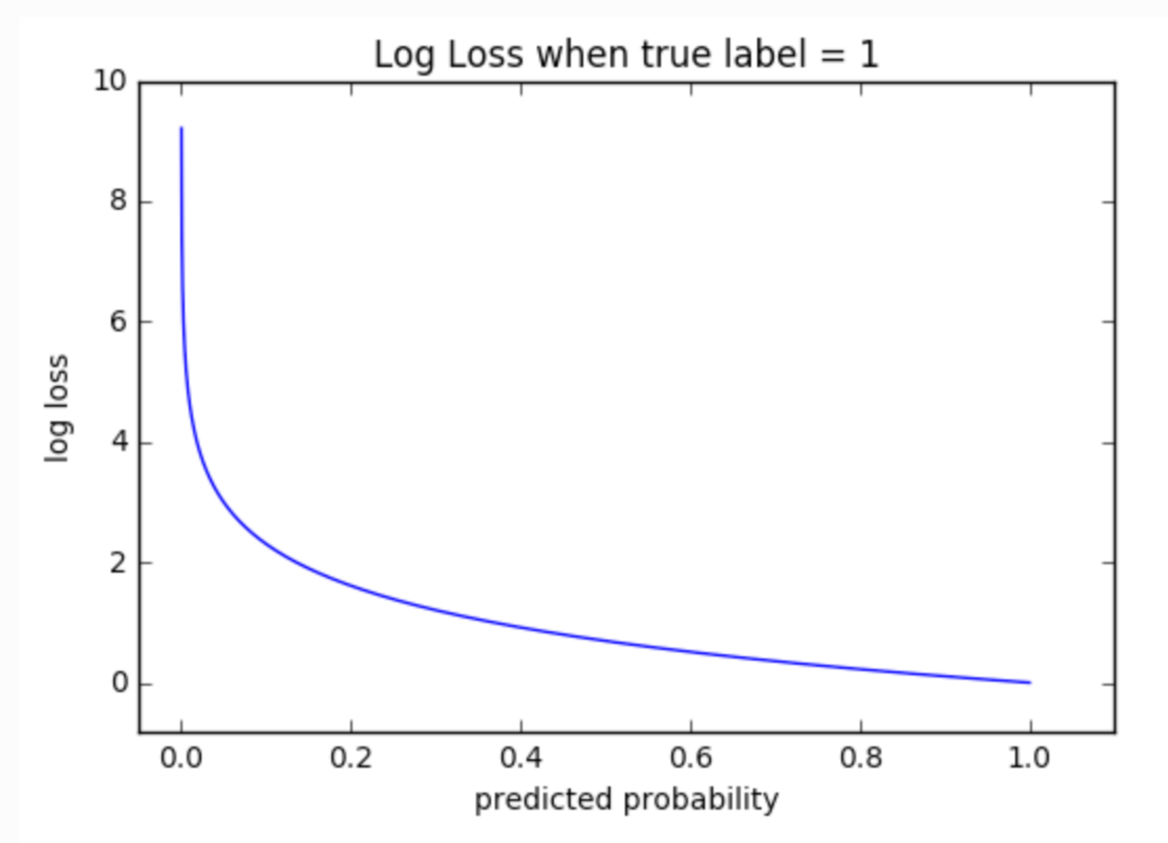
- We've seen *cross-entropy loss* (or *log loss*) over C classes before
    - Measures how similar the actual and predicted probability distributions are
    - Compute cross-entropy $H(y, \hat{y})$ between true y and predicted $\hat{y}$
    - Sum up over all training samples

$$H(y, \hat{y}) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

- For binary classification, this simplifies to

$$-\sum_{c=0,1} y_c \log(\hat{y}_c) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

Cross-entropy loss



For more control, you can explictly create the optimizer, loss, and metrics:

```
from keras import optimizers
from keras import losses
from keras import metrics
model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

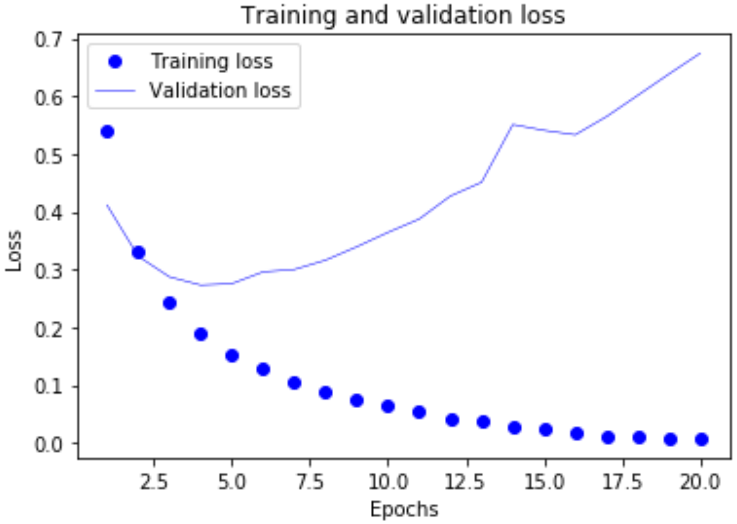
Model selection

- How many epochs do we need for training?
- Take a validation set of 10,000 samples from the training set
- Train the neural net and track the loss after every iteration on the validation set
    - This is returned as a `History` object by the `fit()` function
- We start with 20 epochs in minibatches of 512 samples

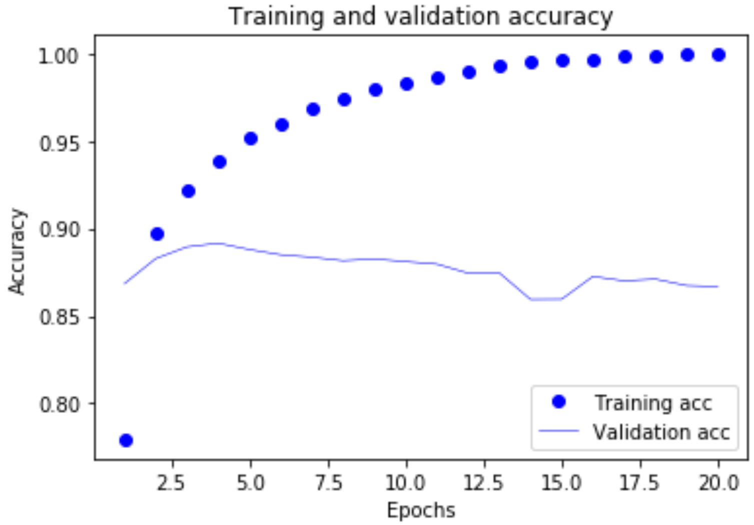
```
x_val, partial_x_train = x_train[:10000], x_train[10000:]
y_val, partial_y_train = y_train[:10000], y_train[10000:]
history = model.fit(partial_x_train, partial_y_train,
                    epochs=20, batch_size=512, verbose=0,
                    validation_data=(x_val, y_val))
```

We can now retrieve visualize the loss on the validation data

- The training loss keeps decreasing, due to gradient descent
- The validation loss peaks after a few epochs, after which the model starts to overfit



We can also visualize the accuracy, with similar findings



Early stopping

One simple technique to avoid overfitting is to use the validation set to 'tune' the optimal number of epochs

- In this case, we could stop after 4 epochs

```
model.fit(x_train, y_train, epochs=4, batch_size=512, verbose=0)
result = model.evaluate(x_test, y_test)
```

```
25000/25000 [==============================] - 4s 149us/step
Loss: 0.4732, Accuracy:  0.8586
```

Predictions

Out of curiosity, let's look at a few predictions:

```
Review 0:  ? please give this one a miss br br ? ? and the rest of the cas
t rendered terrible performances the show is flat flat flat br br i don't
know how michael madison could have allowed this one on his plate he almos
t seemed to know this wasn't going to work out and his performance was qui
te ? so all you madison fans give this a miss
Predicted positiveness:  [0.017]

Review 16:  ? from 1996 first i watched this movie i feel never reach the
end of my satisfaction i feel that i want to watch more and more until now
my god i don't believe it was ten years ago and i can believe that i almos
t remember every word of the dialogues i love this movie and i love this n
ovel absolutely perfection i love willem ? he has a strange voice to spell
the words black night and i always say it for many times never being bored
i love the music of it's so much made me come into another world deep in m
y heart anyone can feel what i feel and anyone could make the movie like t
his i don't believe so thanks thanks
Predicted positiveness:  [0.887]
```

Takeaways

- Neural nets require a lot of preprocessing to create tensors
- Dense layers with ReLU activation can solve a wide range of problems
- Binary classification can be done with a Dense layer with a single unit, sigmoid activation, and binary cross-entropy loss
- Neural nets overfit easily
- Many design choices have an effect on accuracy and overfitting. Try:
    - 1 or 3 hidden layers
    - more or fewer hidden units (e.g. 64)
    - MSE loss instead of binary cross-entropy
    - `tanh` activation instead of `ReLU`

Wrapping Keras models as scikit-learn estimators

- Model selection can be tedious in pure Keras
- We can use all the power of scikit-learn by wrapping Keras models

```python
from keras.wrappers.scikit_learn import KerasClassifier, KerasRegressor
clf = KerasClassifier(model)
param_grid = {'epochs': [1, 5, 10],  # epochs is a fit parameter
              'hidden_size': [32, 64, 256]} # this is a make_model parameter
grid = GridSearchCV(clf, param_grid=param_grid, cv=3)
grid.fit(x_train, y_train)
```

```
Out[11]:  GridSearchCV(cv=3, error_score='raise-deprecating',
                 estimator=<keras.wrappers.scikit_learn.KerasClassifier object at 0
          x13dca8f28>,
                 fit_params=None, iid='warn', n_jobs=None,
                 param_grid={'epochs': [1, 5, 10], 'hidden_size': [32, 64, 256], 'v
          erbose': [0]},
                 pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                 scoring=None, verbose=0)
```

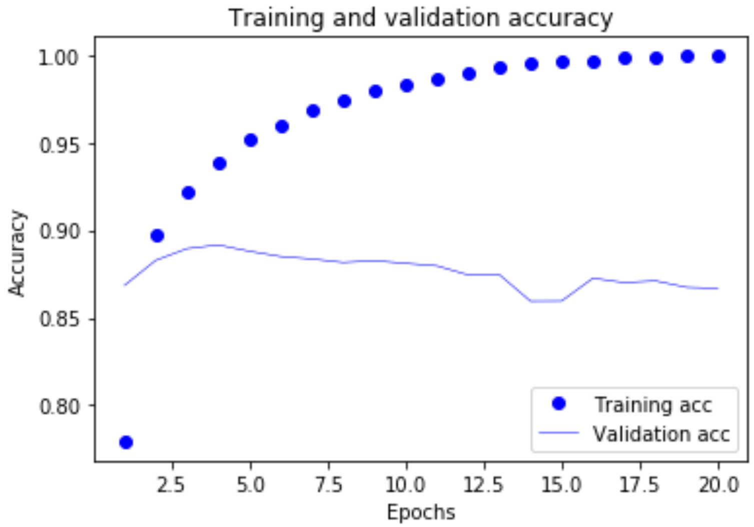
Grid search results

Out[12]:

		mean_test_score	mean_train_score
param_epochs	**param_hidden_size**		
1	**32**	0.89	0.94
	64	0.89	0.93
	256	0.88	0.93
5	**32**	0.88	0.98
	64	0.88	0.97
	256	0.88	0.97
10	**32**	0.87	0.99
	64	0.87	0.99
	256	0.87	0.99

Go a bit deeper: 3 hidden layers

- Not really worth it, very similar results



Multi-class classification (topic classification)

- Dataset: 11,000 news stories, 46 topics
    - Included in Keras, with a 50/50 train-test split
- Each row is one news story, with only the 10,000 most frequent words retained
- Each word is replaced by a *word index* (word ID)

```
News wire:  ? ? ? said as a result of its december acquisition of space co
it expects earnings per share in 1987 of 1 15 to 1 30 dlrs per share up fr
om 70 cts in 1986 the company said pretax net should rise to nine to 10 ml
n dlrs from six mln dlrs in 1986 and rental operation revenues to 19 to 22
mln dlrs from 12 5 mln dlrs it said cash flow per share this year should b
e 2 50 to three dlrs reuter 3
Encoded:  [1, 2, 2, 8, 43, 10, 447, 5, 25, 207, 270, 5, 3095, 111, 16, 36
9, 186, 90, 67, 7]
Topic:  3
```

Preparing the data

- We have to vectorize the data again (using one-hot-encoding)
- We have to vectorize the labels as well, also using one-hot-encoding
    - We can use Keras' `to_categorical` again
    - This yields a vector of 46 floats (0/1) for every sample

```
from keras.utils.np_utils import to_categorical
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Building the network

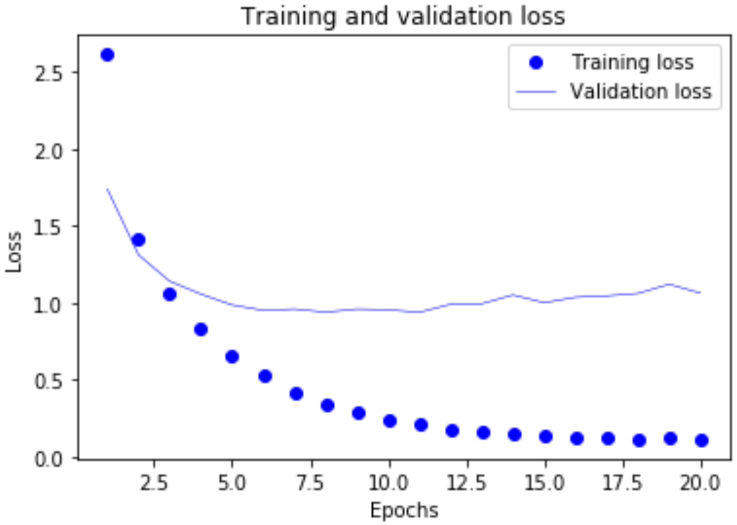
- *Information bottleneck*: Every layer can `drop` some information, which can never be recovered by subsequent layers
- 16 hidden units may be too limited to learn 46 topics, hence we use 64 in each layer
- The output layer now needs 46 units, one for each topic
    - We use `softmax` activation for the output to get probabilities]
- The loss function is now `categorical_crossentropy`

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

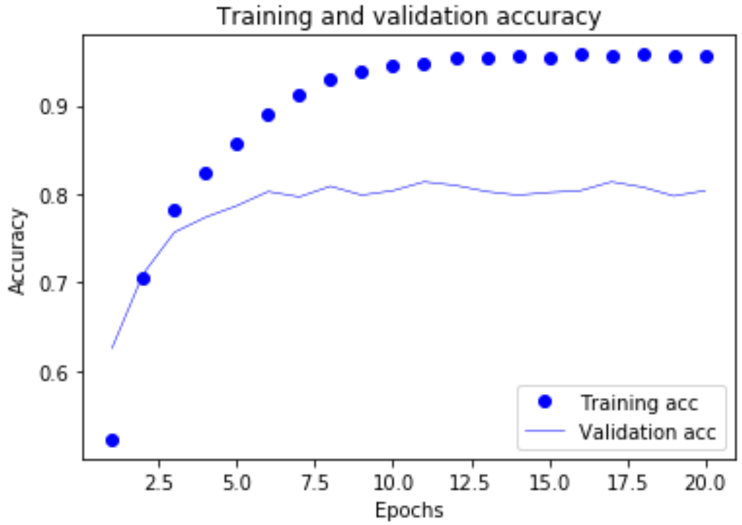
Model selection

- Take a validation set from the training set
- Fit again with 20 epochs

Loss curve:


Training and validation loss
● Training loss
Validation loss
Loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Accuracy curve. Overfitting starts after about 8 epochs



Retrain with early stopping after 8 epochs and validate

```
 model.fit(partial_x_train, partial_y_train, epochs=8, batch_size=512, ve
 rbose=0,)
 result = model.evaluate(x_test, one_hot_test_labels)
```

```
2246/2246 [==============================] - 0s 161us/step
Loss: 1.3851, Accuracy: 0.7836
```

Information bottleneck

- What happens if we create an information bottleneck on purpose
    - Use only 4 hidden units in the second layer
- Accuracy drops dramatically!
- We are trying to learn 46 separating hyperplanes from a 4-dimensional representation
    - It manages to save a lot of information, but also loses a lot

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

```
2246/2246 [==============================] - 0s 120us/step
Loss: 2.2392, Accuracy: 0.6932
```

Takeaways

- For a problem with C classes, the final Dense layer needs C units
- Use `softmax` activation and `categorical_crossentropy` loss
- Information bottleneck: when classifying many classes, the hidden layers should be large enough
- Many design choices have an effect on accuracy and overfitting. Try:
    - 1 or 3 hidden layers
    - more or fewer hidden units (e.g. 128)

Regression

- Dataset: 506 examples of houses and sale prices (Boston)
    - Included in Keras, with a 1/5 train-test split
- Each row is one house price, described by numeric properties of the house and neighborhood
- Small dataset, non-normalized features

Preprocessing

- Neural nets work a lot better if we normalize the features first.
- Keras has no built-in support so we have to do this manually (or with scikit-learn)
    - Again, be careful not to look at the test data during normalization

```
mean, std = train_data.mean(axis=0), train_data.std(axis=0)
train_data -= mean
train_data /= std
test_data -= mean
test_data /= std
```

Building the network

- This is a small dataset, so easy to overfit
    - We use 2 hidden layers of 64 units each
- Use smaller batches, more epochs
- Since we want scalar output, the output layer is one unit without activation
- Loss function is Mean Squared Error (bigger penalty)
- Evaluation metric is Mean Absolute Error (more interpretable)
- We will also use cross-validation, so we wrap the model building in a function, so that we can call it multiple times

```python
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Cross-validation

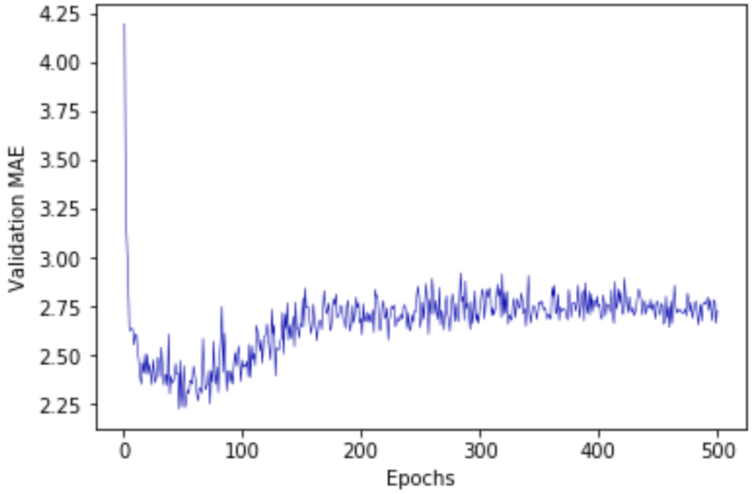
- Keras does not have support for cross-validation
- Luckily we can wrap a Keras model as a scikit-learn estimate
- We can also implement cross-validation ourselves (see notebook)
- Generally speaking, cross-validation is tricky with neural nets
    - Some fold may not converge, or fluctuate on random initialization

```
clf = KerasClassifier(build_model)
score = cross_val_score(clf, train_data, train_targets, scoring='neg_mean_absolute_error', cv=4, fit_params={'epochs': 100, 'batch_size':1, 'verbose':0});
```

```
MAE:  2.41963392496109
```

Train for longer (500 epochs) and keep track of loss after every epoch (see code in notebook)

The model starts overfitting after epoch 80



Retrain with optimized number of epochs

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
```

```
102/102 [==============================] - 0s 191us/step
MAE:  3.202784299850464
```

Takeaways

- Regression is usually done using MSE loss and MAE for evaluation
- Input data should always be scaled (independent from the test set)
- Small datasets:
    - Use cross-validation
    - Use simple (non-deep) networks
    - Smaller batches, more epochs

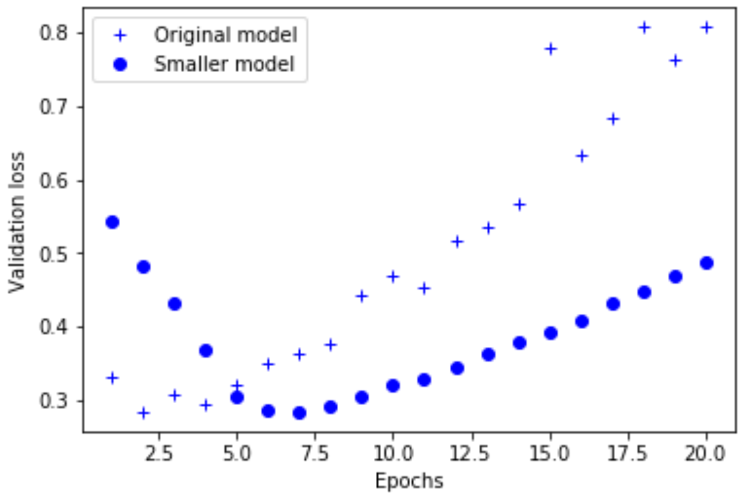
Regularization: build smaller networks

- The easiest way to avoid overfitting is to use a simpler model
- The number of learnable parameters is called the model *capacity*
- A model with more parameters has a higher *memorization capacity*
    - The entire training set can be `stored` in the weights
    - Learns the mapping from training examples to outputs
- Forcing the model to be small forces it to learn a compressed representation that generalizes better
    - Always a trade-off between too much and too little capacity
- Start with few layers and parameters, incease until you see diminisching returns

Let's try this on our movie review data, with 4 units per layer

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

The smaller model starts overfitting later than the original one, and it overfits more *slowly*



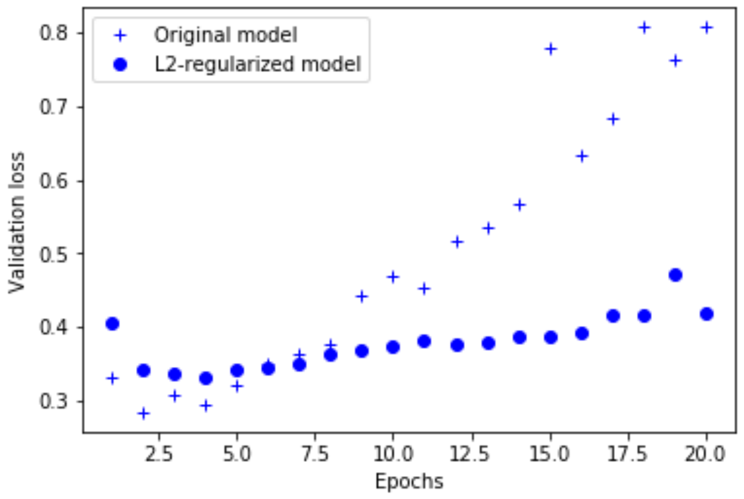
Regularization: Weight regularization

- As we did many times before, we can also add weight regularization to our loss function
- L1 regularization: leads to *sparse networks* with many weights that are 0
- L2 regularization: leads to many very small weights
    - Also called *weight decay* in neural net literature
- In Keras, add `kernel_regularizer` to every layer

```
from keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))
```

L2 regularized model is much more resistant to overfitting, even though both have the same number of parameters



You can also try L1 loss or both at the same time

```python
from keras import regularizers

# L1 regularization
regularizers.l1(0.001)

# L1 and L2 regularization at the same time
regularizers.l1_l2(l1=0.001, l2=0.001)
```

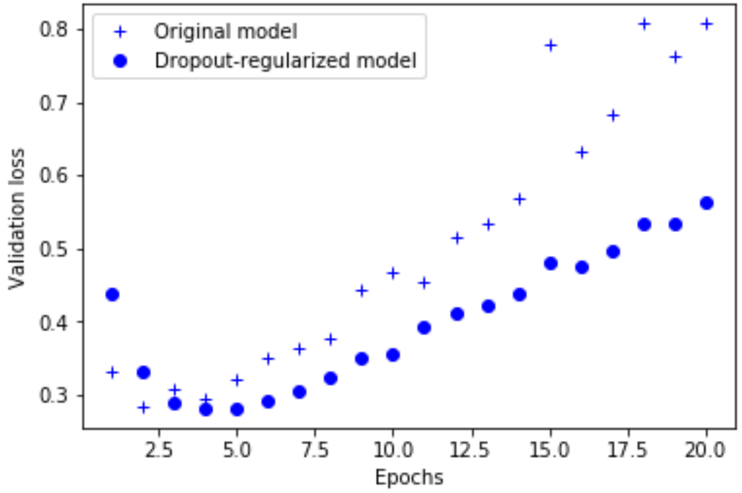
Regularization: dropout

- One of the most effective and commonly used regularization techniques
- Randomly set a number of outputs of the layer to 0
- Idea: break up accidental non-significant learned patterns
- *Dropout rate*: fraction of the outputs that are zeroed-out
    - Usually between 0.2 and 0.5
- At test time, nothing is dropped out, but the output values are scaled down by the dropout rate
    - Balances out that more units are active than during training
- In Keras: add `Dropout` layers between the normal layers

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

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['acc'])
```

Dropout finds a better model, and overfits more slowly as well



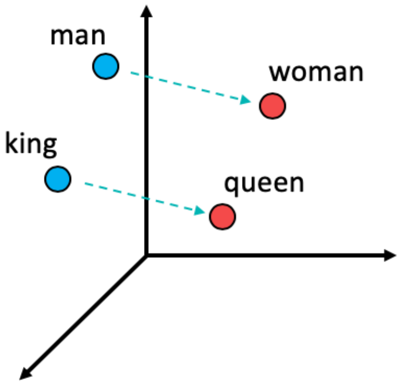
Regularization recap

- Get more training data
- Reduce the capacity of the network
- Add weight regularization
- Add dropout
- Either start with a simple model and add capacity
- Or, start with a complex model and then regularize by adding weight regularization and dropout

Word Embeddings

- An embedding maps each word to a point in an n-dimensional space (e.g. 300 values), so that similar words are close to each other
- There are different ways to learn word embeddings. Most common are:
    - Word2Vec: Learn encoding based on the words that are typically in a window around the word
        - Encoding is learned using a 1-layer neural net
    - GloVe (Global Vector): Count co-occurrences of words in a matrix
        - Use a low-rank approximation to get a condensed vector representation
    - FastText: learns embedding for n-grams rather than complete words
        - Can also produce embeddings for new words
    - BERT, ELMO,...: learn a context-dependent embedding: the some word has a different embedding depending on the sentence it appears in

Word embeddings: Word2Vec and GloVe



woman

$$=\begin{bmatrix}0.29\\3.11\\1.29\\\cdots\\\cdots\\\cdots\\2.74\\1.49\\3.49\end{bmatrix}$$

300 values

Word2Vec (prediction)

I like playing football with my friends

This is the tallest tree I have ever seen

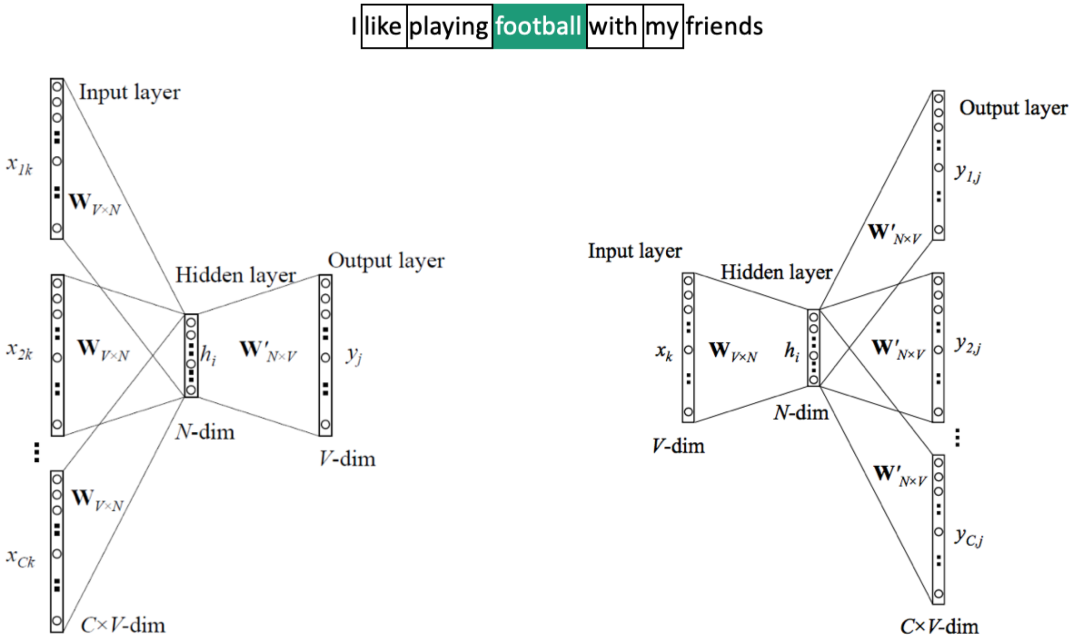
I have trained my dog well

GloVe (count)

	my	tallest	trained
football	1	0	0
tree	0	1	0
dog	1	0	1

Word2Vec

- Continuous bag of words (CBOW): train model to predict word given context, use weights of last layer as embedding
- Skip-Gram: train model to predict context given word, use weights of first layer as embedding (better for large corpora)



CBOW

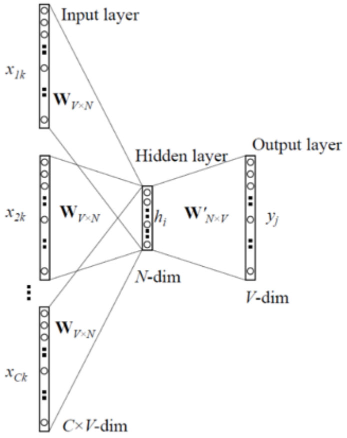
predict word given context

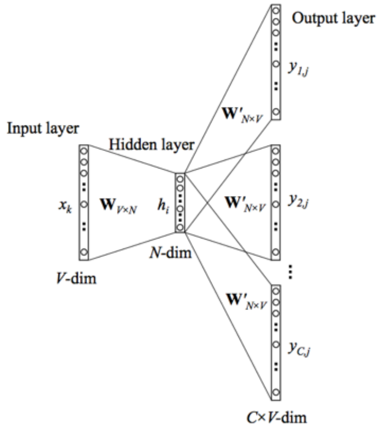
Skip-gram

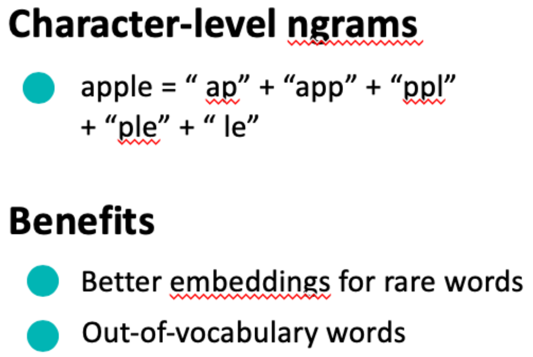
predict context given word

FastText

Like CBOW or SkipGram, but using n-grams instead of words







Character-level ngrams

- apple = “ ap” + “app” + “ppl” + “ple” + “ le”

Benefits

- Better embeddings for rare words
- Out-of-vocabulary words

BERT, ELMO, GPT,...

- Use transformer networks to learn contextual embeddings
- Devlin et al. (2018) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

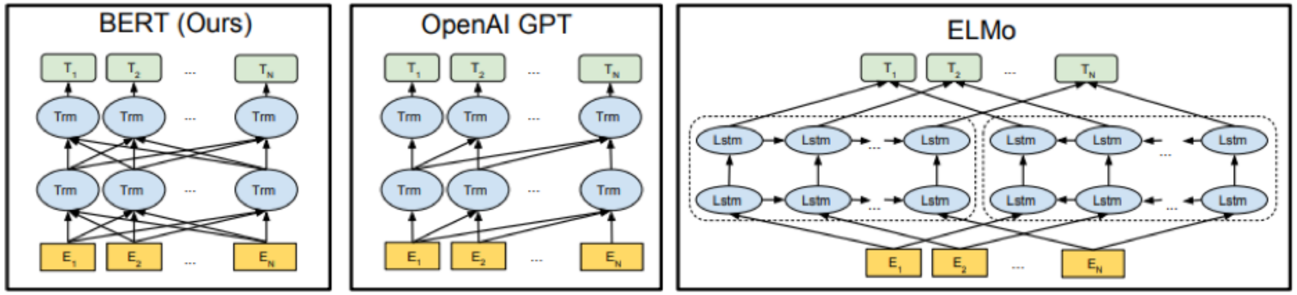


Figure 1: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTM to generate features for downstream tasks. Among three, only BERT representations are jointly conditioned on both left and right context in all layers.

Using Embeddings in Keras

- Uses a pre-trained embedding, obtained using GloVe
- See a complete example here (https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)

```
max_length = 20 # pad documents to a maximum number of words
vocab_size = 10000 # vocabulary size
embedding_length = 300 # vocabulary size
# define the model
model = Sequential()
model.add(Embedding(vocab_size, embedding_length, input_length=max_lengt
h))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[
'accuracy'])
# summarize the model
print(model.summary())
```