Neural Nets II

Elliott Ash

Text Data Course, Bocconi 2018

Tuning NN Hyperparameters

- Number of hidden layers:
  - having a single hidden layer will generally give decent results.
    - more layers with fewer neurons can recover hierarchical relations and complex functions
    - for text classification, try one or two hidden layers as a baseline.
- Number of neurons:
  - a common practice is to set neuron counts like a funnel, with fewer and fewer neurons at each level
  - or just pick 150 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Activation functions:
  - ReLU works well for hidden layers
  - softmax is good for the output layer in classification tasks

Xavier and He Initialization

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Connection weights should be initialized randomly according to a uniform distribution or normal distribution, as indicated in the table (see Geron Chapter 11).

```
model.add(Dense(64, kernel_initializer='he_normal'))
model.add(Dense(64, kernel_initializer='he_uniform'))
```

Other Activation Functions

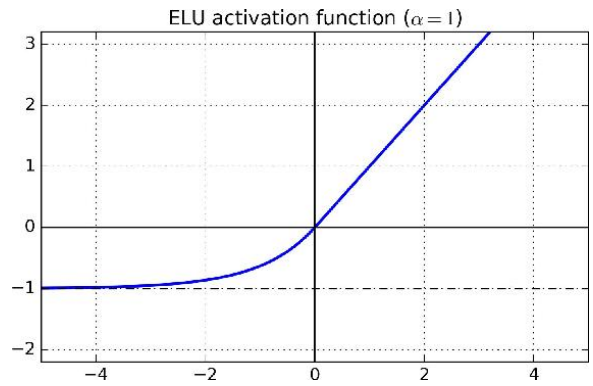
- Leaky ReLU

$$\max(\alpha z, z)$$

where α is set to a small number, such as .01, or learned in training.

- Exponential linear unit

$$\text{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



- In general, ELU has had the best performance so far, but it is slower than ReLU.

Batch normalization

- Another trick to speed up training:
  - in between layers, zero-center and normalize the inputs to variance one.
  - normally done before a non-linear activation function

```
from keras.layers.normalization import BatchNormalization
model.add(Dense(64, use_bias=False))
model.add(BatchNormalization())
model.add(Activation('elu'))
```

Regularization for Sparse Models

- As with linear models, neural network parameters can be regularized with an L1 and/or L2 penalty to push weak neurons to zero and produce a sparse model.

```
from keras.regularizers import l1, l2, l1_l2
model.add(Dense(64,
                kernel_regularizer=l2(0.01),
                activity_regularizer=l1(0.01)))
model.add(Dense(64,
                kernel_regularizer=l1_l2(l1=0.01,l2=.01),
                activity_regularizer=l1_l2(l1=0.01,l2=.01)))
```

Dropout

- An elegant regularization technique:
  - at every training step, every neuron has some probability (typically 0.5) of being temporarily dropped out, so that it will be ignored at this step.
  - after training, neurons dont get dropped any more.
- Neurons trained with dropout:
  - cannot co-adapt with neighboring neurons and must be independently useful.
  - cannot rely excessively on just a few input neurons; they have to pay attention to all input neurons.
    - makes the model less sensitive to slight changes in the inputs.
- If a model is over-fitting, increase dropout. Dropout can be higher for large layers and lower for small layers.

```
from keras.layers import Dropout
model.add(Dropout(0.5))
```

Optimizers and loss functions

- Choice of optimization algorithm is the topic of active research, which has shown that it can have a big impact on model performance.

```
model.compile(optimizer='adam',loss='binary_crossentropy')
model.compile(optimizer='sgd',loss='binary_crossentropy')
```

- A good starting choice is Adam (adaptive moment estimation), which is fast and usually works well. For robustness, can also try SGD.
- Loss functions:

Prediction Task	Loss Function to Use
binary classification	`binary_crossentropy`
multi-class classification	`categorical_crossentropy`
regression	`mean_squared_error`

Early stopping

- A popular/efficient regularization method is to continually evaluate your model at regular intervals, and then to stop training when the test-set accuracy starts to decrease.

```
from keras.callbacks import EarlyStopping
earlystop = EarlyStopping(monitor='val_acc',
                          min_delta=0.0001,
                          patience=5,
                          verbose=1,
                          mode='auto')
callbacks_list = [earlystop]
model.fit(X, Y, callbacks=callbacks_list,
          validation_split=0.2)
```

Practical Guidelines

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Batch Training with Large Data

- If data sets don't fit in memory, one can load the data in batches from disk.

```
from numpy import memmap
X_mm = memmap('X.pkl',shape=(32567, 472))

model.fit(X_mm, Y, batch_size=128,
          epochs=100,
          callbacks=callbacks_list,
          validation_split=0.2)
```

- can also continuously update a saved model.

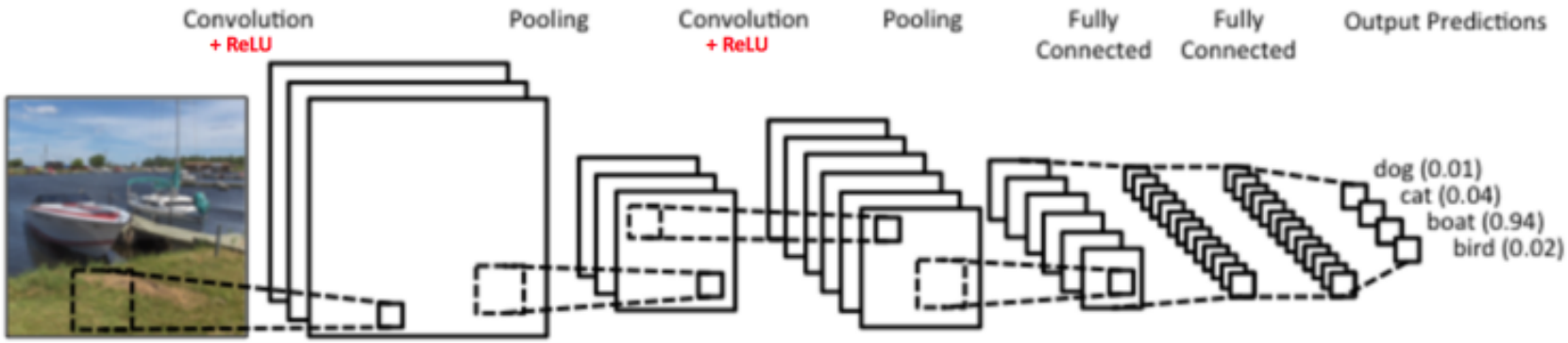
Grid search for model choice

- The flexibility of DNNs is a blessing and a curse.
  - in general, one should make a complex model that allows regularization.
- But still, there are many choices to be made.
  - to choose the number of hidden layers, for example, one can use cross-validation grid search (as we did with standard scikit-learn models).

Grid search for model choice (code)

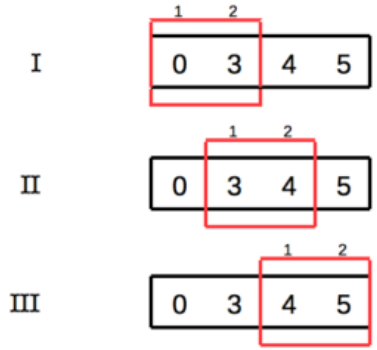
```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

# instantiate KerasClassifier with build function
def create_model(hidden_layers=1):
    model = Sequential()
    model.add(Dense(16, input_dim=num_features))
    for i in range(hidden_layers):
        model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    return model
clf = KerasClassifier(create_model)

# set up grid search CV to select number of hidden layers
params = {'hidden_layers' : [0,1,2,3]}
grid = GridSearchCV(clf, param_grid=params)
grid.fit(X,Y)
grid.best_params_
```

Convolutional neural nets



- CNNS are a special category of deep neural nets that have been especially effective at image classification.

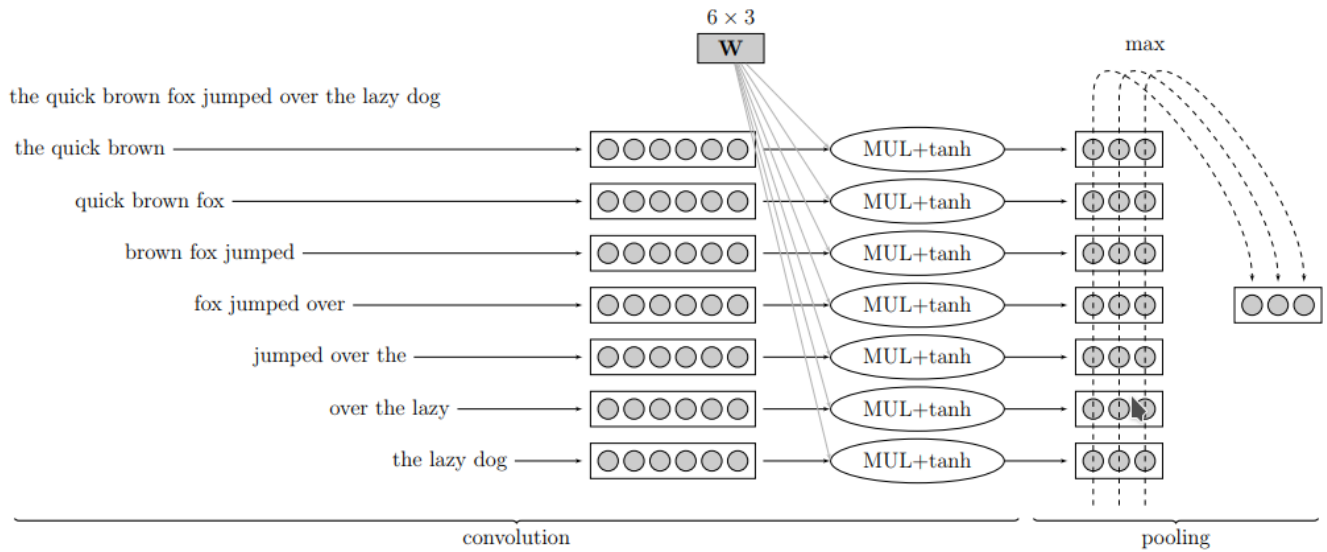
Sequence Convolution



- CNN's generate filters, such as the $\{1, 2\}$ here, and slides the filters across the input sequence.
- At each window, take the dot product:
  - `[0 3] * [1 2] = 6, [3 4] * [1 2] = 11, [4 5] * [1 2] = 14`
  - output $= \{6, 11, 14\}$
- CNN learns the weights for the filter $\{w_1, w_2\}$, to try to match the output ($w_1 = 1$ and $w_2 = 2$ in this example).
  - complicated CNNs have more filters, and different filter window sizes.

Convolutional Neural Nets for Text

- CNNs are useful for classification tasks where we expect to find strong local clues regarding class membership, but where the clues could appear in different places in the input documents.



- The sliding window of length n learns to identify informative n-grams.

Johnson and Zhang 2014

- Linear models tend to rely on single words and just a few n-grams. CNNs recover the predictive power of longer phrases:

N1	completely useless ., return policy .
N2	it won't even, but doesn't work
N3	product is defective, very disappointing !
N4	is totally unacceptable, is so bad
N5	was very poor, it has failed
P1	works perfectly !, love this product
P2	very pleased !, super easy to, i am pleased
P3	'm so happy, it works perfect, is awesome !
P4	highly recommend it, highly recommended !
P5	am extremely satisfied, is super fast

Table 5: Examples of predictive text regions in the training set.

were unacceptably bad, is abysmally bad, were universally poor, was hugely disappointed, was enormously disappointed, is monumentally frustrating, are endlessly frustrating
best concept ever, best ideas ever, best hub ever, am wholly satisfied, am entirely satisfied, am incredicbly satisfied, 'm overall impressed, am awfully pleased, am exceptionally pleased, 'm entirely happy, are acoustically good, is blindingly fast,

Table 6: Examples of text regions that contribute to prediction. They are from the *test set*, and they did *not* appear in the training set, either entirely or partially as bi-grams.

CNN's in Code

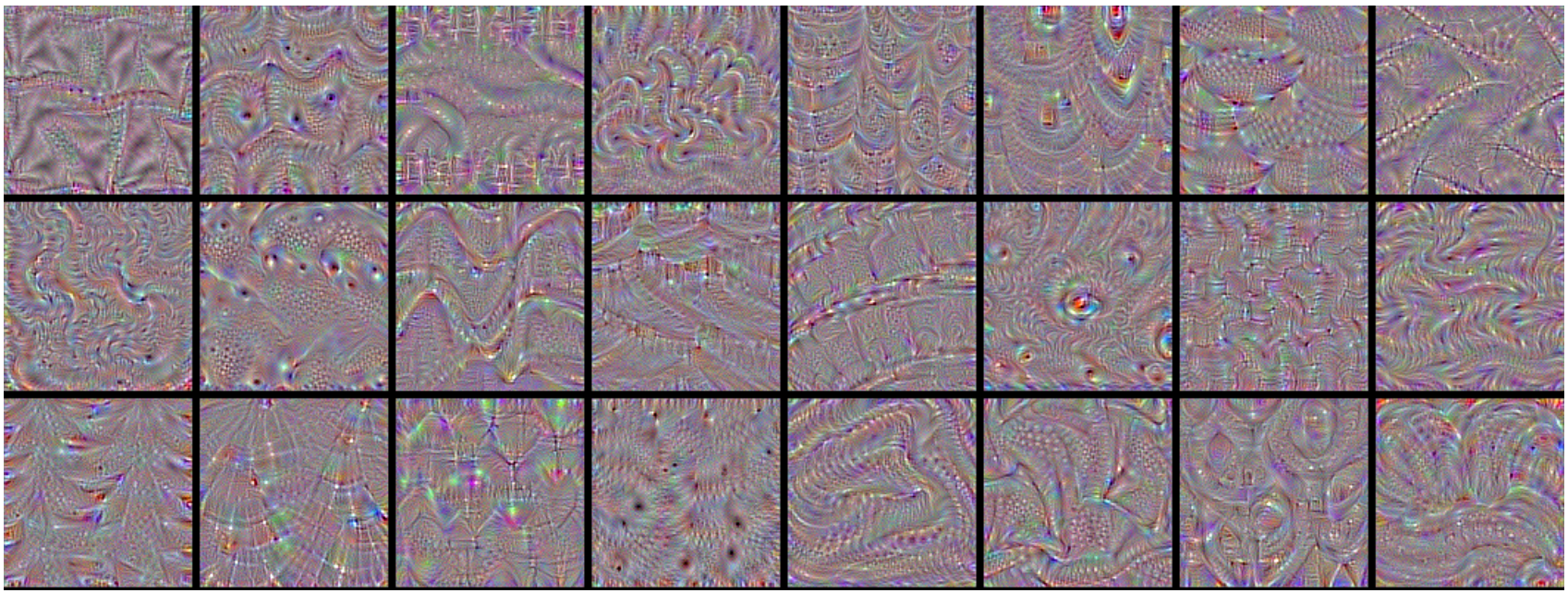
```
from keras.layers import Conv1D, GlobalMaxPooling1D
model = Sequential()
model.add(Conv1D(input_shape=(69,100),
                 filters=250,
                 kernel_size=3)) # trigrams
model.add(GlobalMaxPooling1D())
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

- The Google Text Classification Guide recommends the SeparableConv1D class (rather than Conv1D); gives similar performance but faster to train.
  - See `https://developers.google.com/machine-learning/guides/text-classification/step-4#build_sequence_model_option_b`

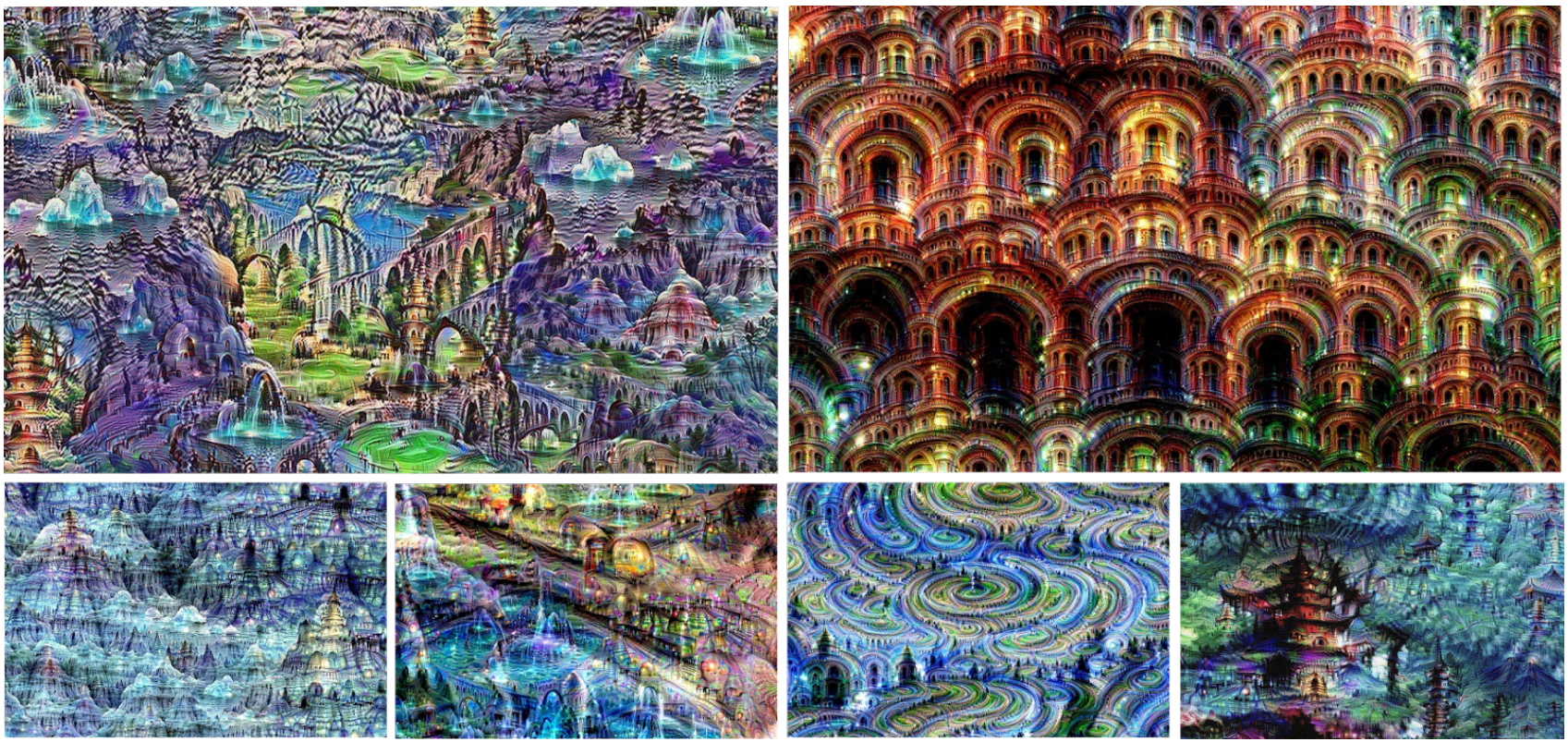
Interrogating CNN Filters

```
# weights on first trigram filter
first_filter = model.get_weights()[0][:,:,0]
first_filter.shape

# highest activating trigram
trigram = np.argmax(np.abs(first_filter),axis=1)
' '.join([id2word[i] for i in trigram])
```

How CNNs see the world

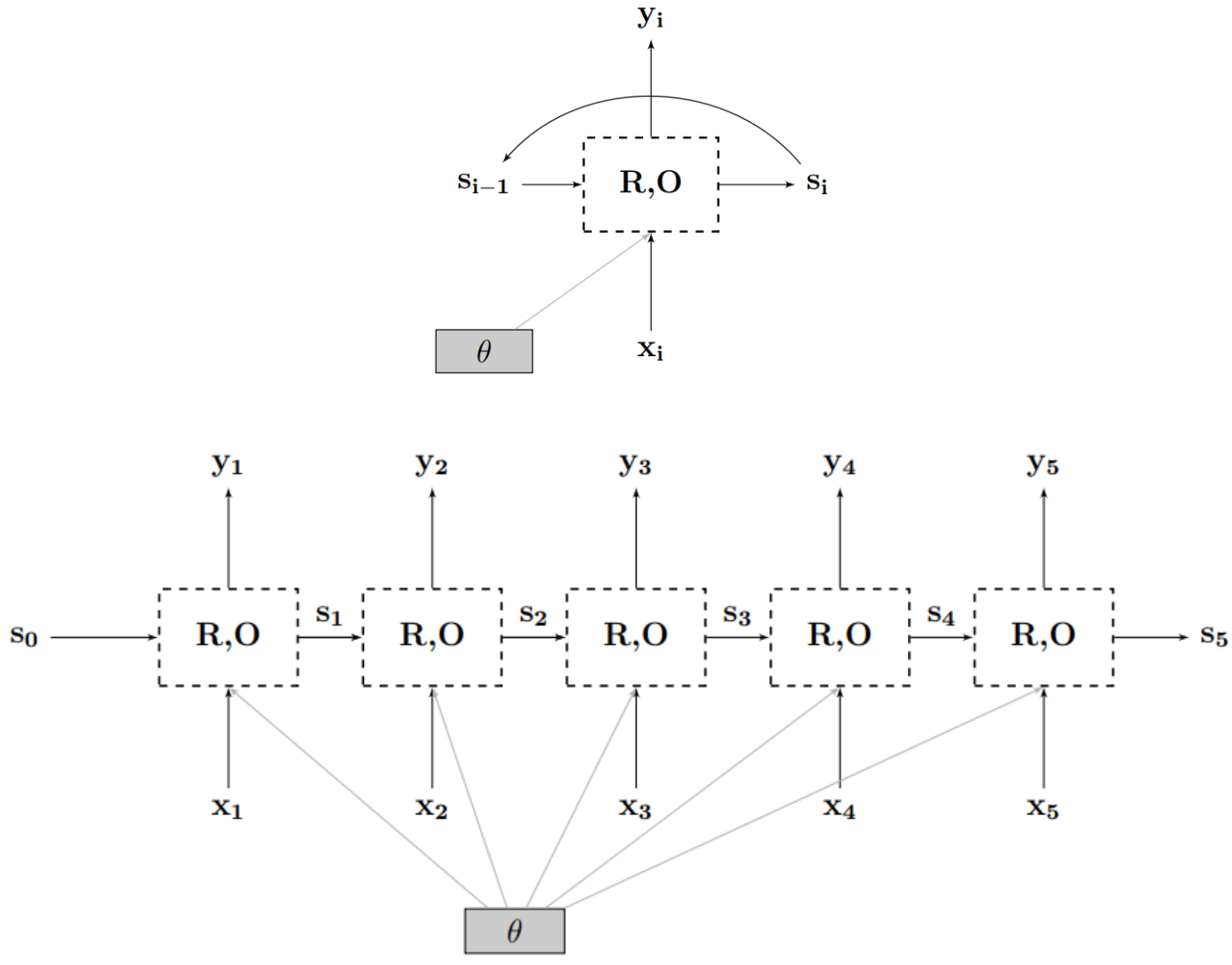


How CNNs see the world



From vectors to sequences

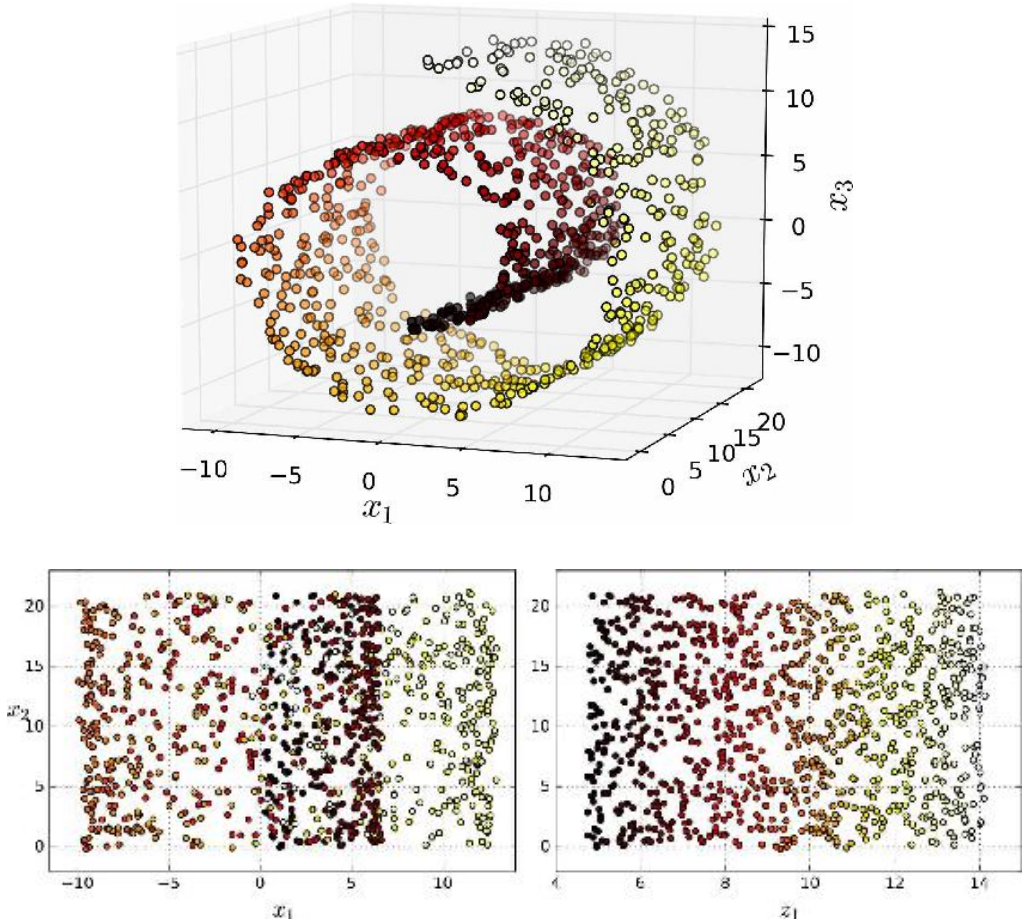
- The models we have looked at so far took inputs of fixed dimensions across rows.
- RNNs can work with sequences of arbitrary length.
  - they are useful for language tasks such as translation.


y_i
s_{i-1}
R,O
s_i
θ
x_i
y_1
y_2
y_3
y_4
y_5
s_0
R,O
s_1
R,O
s_2
R,O
s_3
R,O
s_4
R,O
s_5
x_1
x_2
x_3
x_4
x_5
θ

RNN Implementation

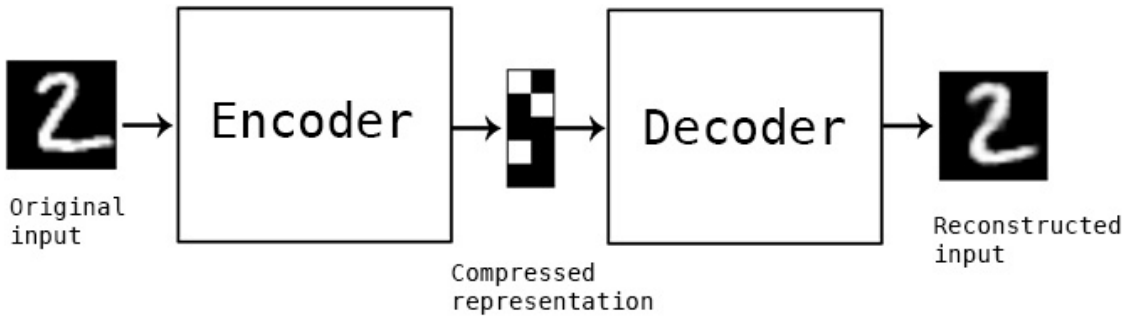
- LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) refer to popular RNN implementations for natural language analysis.
  - Can predicting the next word in a sequence, for example.
  - they don't apply naturally to document-level outcomes – e.g. predicting the citations for a document.
- Can perform sequence to sequence predictions – used for translations.
  - could be used for translating "Republican" language to "Democrat" language?
- A couple of useful tutorials:
  - `http://adventuresinmachinelearning.com/keras-lstm-tutorial/`
  - `https://machinelearningmastery.com/develop-character-based-neural-language-model-keras/`

Remember the Swiss role?



- The dimension reduction process matters: projecting down to two dimensions directly (left panel) might not isolate the variation we are interested in (as done in the right panel, which unrolls the Swiss Roll)

Autoencoders: Domain-specific dimension reduction



- "Autoencoder" refers to a class of deep neural network that performs domain-specific dimension reduction.
  - They learn efficient encodings of the data, which can then be decoded back to a (minimally) lossy representation of the original data.
  - Can also randomly generate new data that looks like the training data.

Autoencoding for data visualization

- For 2D visualization, t-SNE is probably the best algorithm around
  - but quite slow, and typically requires relatively low-dimensional data.
- Good strategy:
  - use an autoencoder to compress your data to relatively low dimension (e.g. 32 dimensions)
  - then use t-SNE for mapping the compressed data to a 2D plane.

Keras autoencoder

```
from keras.layers import Input
data_dims = X.shape[1]
input_img = Input(shape=(data_dims,)) # input placeholder

# encoded: the compressed representation
encoded = Dense(32, activation='relu')(input_img)
# decoded: the lossy reconstruction
decoded = Dense(data_dims, activation='sigmoid')(encoded)

# model maps an input to its reconstruction
from keras.models import Model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta',
                    loss='binary_crossentropy')
```

Fit and Run Dimension Reduction

```
# Fit and validate
autoencoder.fit(X, X,
                epochs=10,
                batch_size=256,
                shuffle=True,
                validation_split=.2)

# decode and re-encode data
Xpred = autoencoder.predict(X)

# Dimension reduction using autoencoder
encoder = Model(input_img, encoded)
X32 = encoder.predict(X)
```