UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Machne Learning**
- Linear and Logistic Regression
- Softmax classification
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

- **Neural Networks**
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions
- Common layers: Conv and Pooling Layers
- CNN Overview

Midterm / Project proposal due (30pts)

- **Working with images**
- Normlization
- Loading Images
- Image formats and manipulation

- **CNN Implementation**
- Training
- Recurrent Neural Network (RNN)
- Project Presentations 1/2

- **Project Presentations**
- Project Presentation 2/2

Final Project (40pts)

Convert Images to TFRecords

```
def write_records_file(dataset, record_location):
    """
    Fill a TFRecords file with the images found in `dataset`
and include their category.

    Parameters
    ----------
    dataset : dict(list)
      Dictionary with each key being a label for the list of
image filenames of its value.
    record_location : str
      Location to store the TFRecord output.
    """
    writer = None

    # Enumerating the dataset because the current index is
used to breakup the files if they get over 100
    # images to avoid a slowdown in writing.
    current_index = 0
    for breed, images_filenames in dataset.items():
        for image_filename in images_filenames:
            if current_index % 100 == 0:
                if writer:
                    writer.close()
```

Convert Images to TFRecords

```
                record_filename = "{record_location}-
{current_index}.tfrecords".format(
                    record_location=record_location,
                    current_index=current_index)

                writer = tf.python_io.TFRecordWriter(re-
cord_filename)
            current_index += 1

            image_file = tf.read_file(image_filename)

            # In ImageNet dogs, there are a few images which
TensorFlow doesn't recognize as JPEGs. This
            # try/catch will ignore those images.
            try:
                image = tf.image.decode_jpeg(image_file)
            except:
                print(image_filename)
                continue

            # Converting to grayscale saves processing and mem-
ory but isn't required.
            grayscale_image = tf.image.rgb_to_grayscale(image)
            resized_image = tf.image.resize_images(gray-
scale_image, 250, 151)
```

```
            # tf.cast is used here because the resized images
are floats but haven't been converted into
            # image floats where an RGB value is between [0,1).
            image_bytes = sess.run(tf.cast(resized_image,
tf.uint8)).tobytes()

            # Instead of using the label as a string, it'd be
more efficient to turn it into either an
            # integer index or a one-hot encoded rank one ten-
sor.
            # https://en.wikipedia.org/wiki/One-hot
            image_label = breed.encode("utf-8")

            example = tf.train.Example(features=tf.train.Fea-
tures(feature={
                'label': tf.train.Fea-
ture(bytes_list=tf.train.BytesList(value=[image_label])),
                'image': tf.train.Fea-
ture(bytes_list=tf.train.BytesList(value=[image_bytes]))
            }))

            writer.write(example.SerializeToString())
    writer.close()

write_records_file(testing_dataset, "./output/testing-images/
testing-image")
write_records_file(training_dataset, "./output/training-images/
training-image")
```

Load Images

- Once the testing and training dataset have been transformed to TFRecord format, they can be read as TFRecords instead of as JPEG images
- The goal is to load the images a few at a time with their corresponding labels

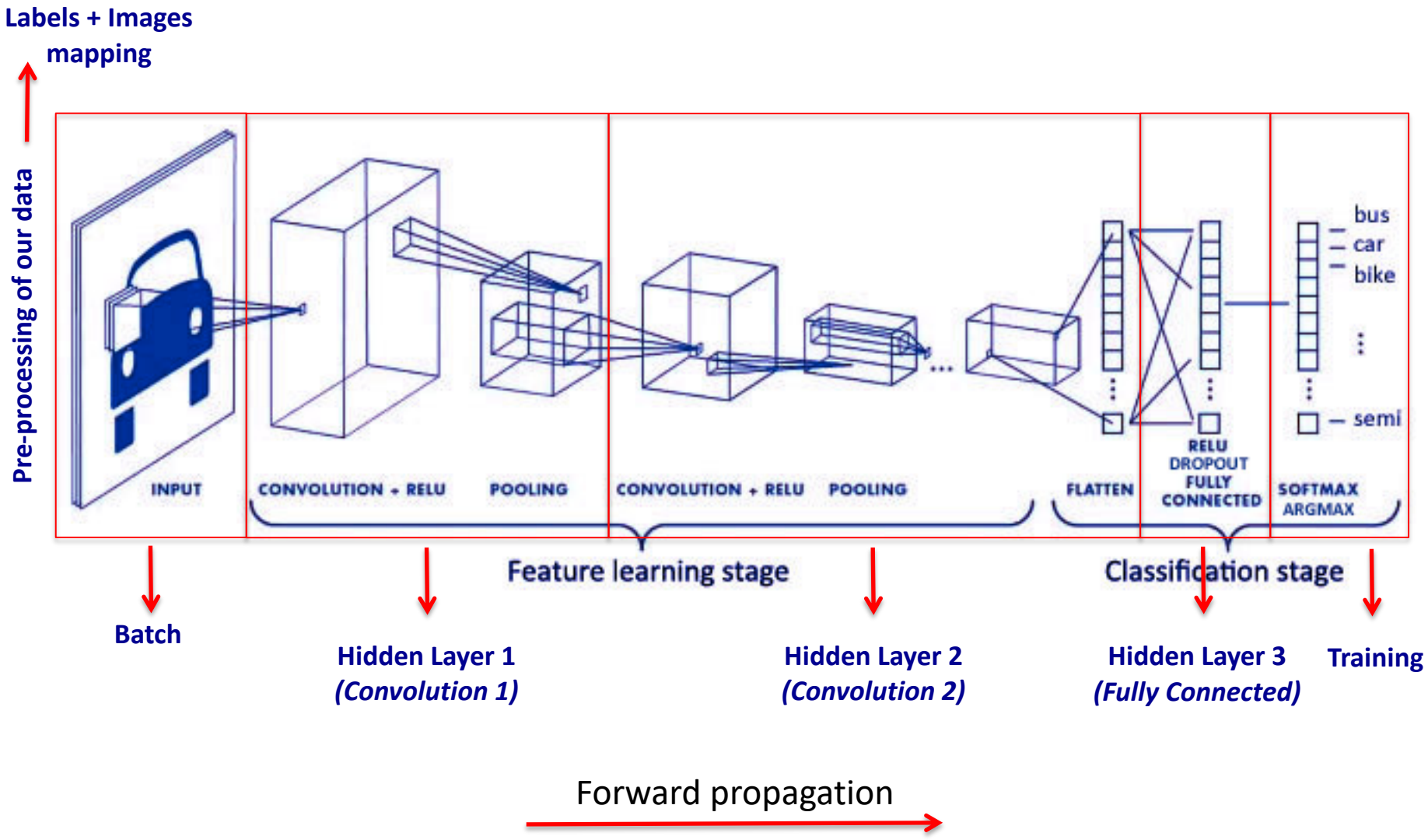
```
filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once("./output/training-images/
*.tfrecords"))
reader = tf.TFRecordReader()
_, serialized = reader.read(filename_queue)
```

Load Images

parse label and image

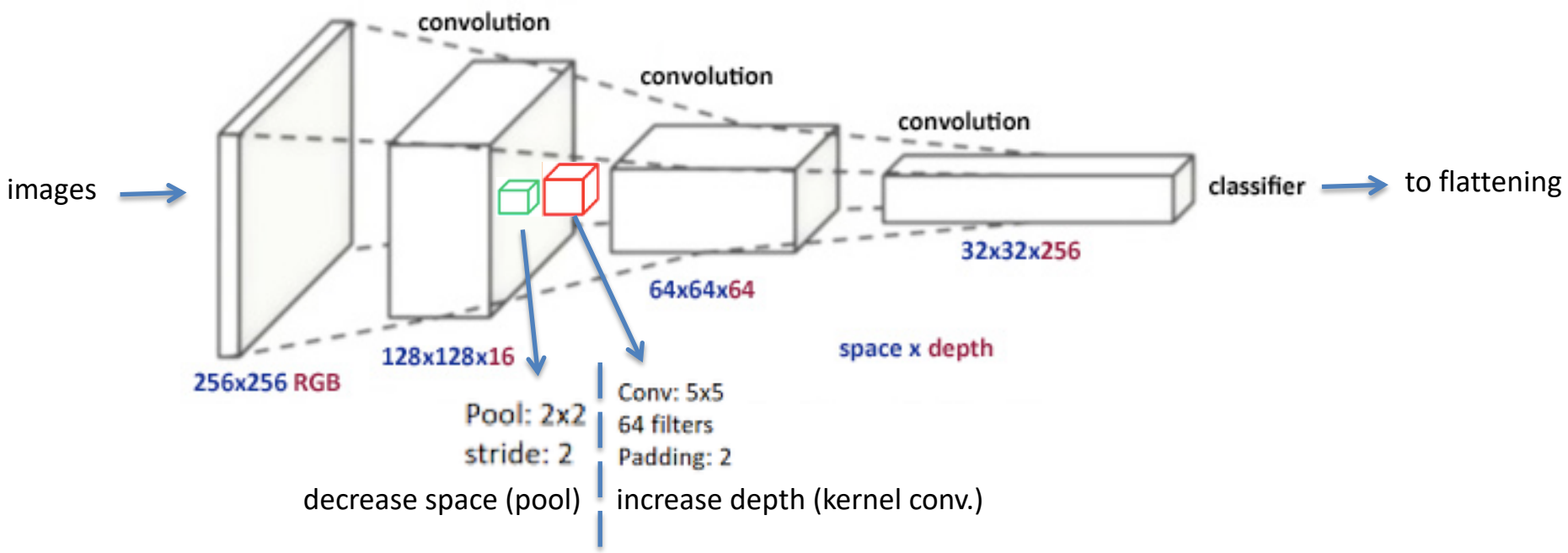
```
features = tf.parse_single_example(
    serialized,
    features={
        'label': tf.FixedLenFeature([], tf.string),
        'image': tf.FixedLenFeature([], tf.string),
    })

record_image = tf.decode_raw(features['image'], tf.uint8)

# Changing the image into this shape helps train and visualize
the output by converting it to
# be organized like an image.
image = tf.reshape(record_image, [250, 151, 1])

label = tf.cast(features['label'], tf.string)

min_after_dequeue = 10
batch_size = 3
capacity = min_after_dequeue + 3 * batch_size
image_batch, label_batch = tf.train.shuffle_batch(
    [image, label], batch_size=batch_size, capacity=capacity,
min_after_dequeue=min_after_dequeue)
```

Model

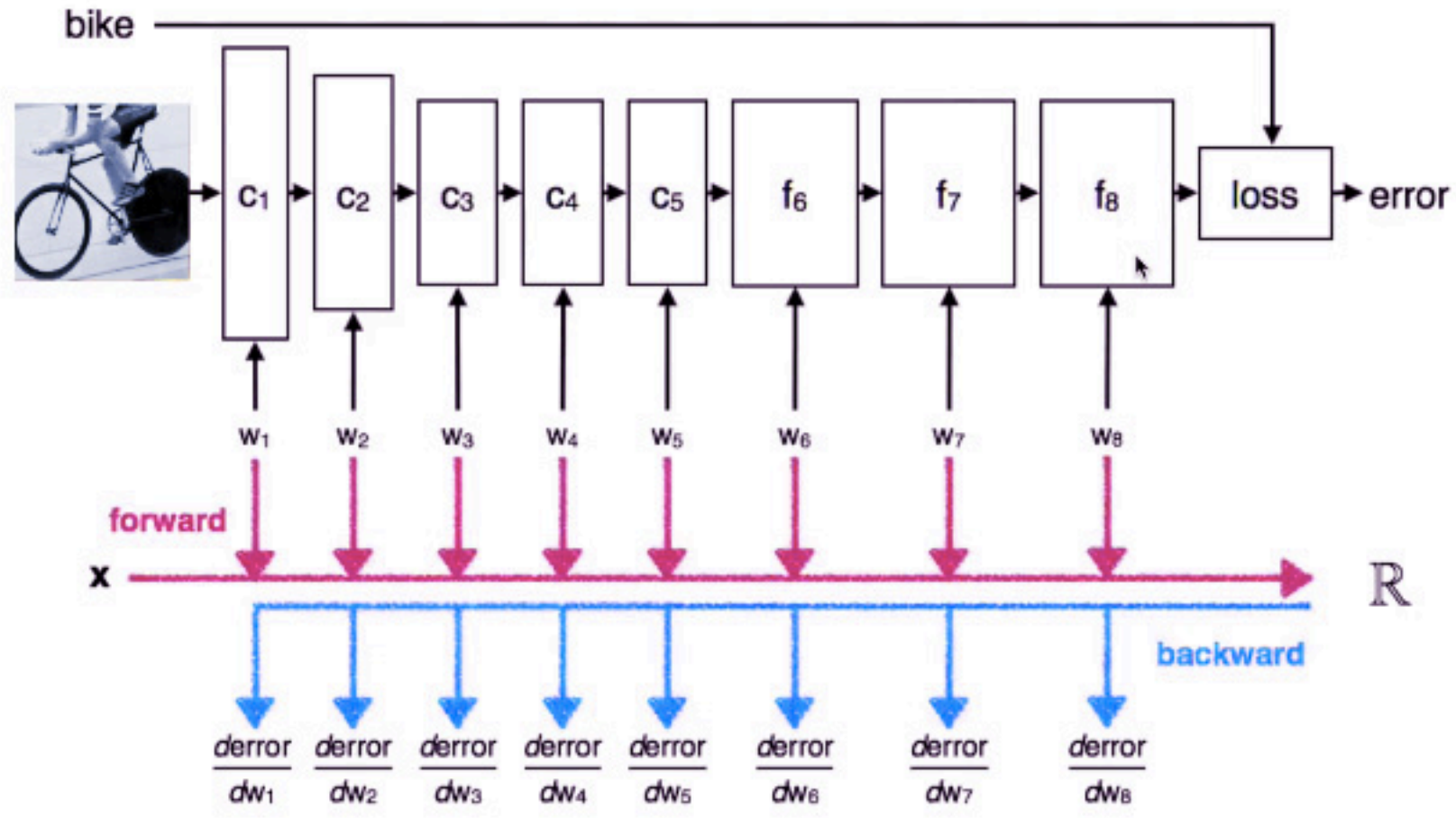

Labels + Images mapping
Pre-processing of our data
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
RELU
DROPOUT
FULLY CONNECTED
SOFTMAX
ARGMAX
bus
car
bike
semi
Feature learning stage
Classification stage
Batch
Hidden Layer 1
(Convolution 1)
Hidden Layer 2
(Convolution 2)
Hidden Layer 3
(Fully Connected)
Training
Forward propagation

CNN clarifications

- In NN matrixes are multiplied (dot product) by other matrixes all the way through the network
- We start with bigger images (space) and after every layer they become more, but smaller (depth)



Model



Backward propagation

CNN clarifications

- These terms mean the same thing in CNN:

  weights = weight matrix = feature matrix = feature map = kernel = filter

- We can specify filters by taking specific parts of images for example or we can use pre-specified filters

- We then convolve these filters with other images and search for suitable overlap, where parts that are not interesting will = 0 and those that are interesting will > 0. Higher values are better !!!

- For that we use ReLU and is called activation. It is applied to every single pixel of the image.

CNN clarifications

- The neurons in each layer perform the same mathematical operation
- We feed it with the data from the previous layer in the network.
- Dropout provides random neuron dropout to prevent overfitting
- Fully connected layer is using softmax for probability (last layer)
- Softmax will give us a probability of our findings [0:1] based on the classes that are more likely to occur.
- We pick the maximum probability class using np.argmax()

CNN clarifications

- We train the network using Gradient Descent = Backpropagation
- The number of hidden layers and neurons is determined experimentally
- Flip a convolutional network and you get a De-convolutional network where you can create an image out of text
- Use 2D and 3D CNN any time we have spatial data (where the space and order matter), such as: images, sounds, text, etc.
- Remember: for regression we use the MSE or RMSE and for classification we use the softmax function

CNN clarifications

- How to improve the performance of a CNN?
  - Vary all hyperparameters such as:
    - Number of inputs (decrease large number of pixels to all images)
    - Number of layers
    - Type, frequency and % of pooling
    - Droput to prevent overfitting
  - Image pre-processing techniques:
    - Use uniform size and aspect ratio for all images
    - Use scaling, cropping and padding of images to make them of equal size
    - Use mean pixel values across all training examples (for each pixel)
    - Image normalization: subtract the mean from each pixel and then dividing the result by the STD
    - Convert images to grayscale (only if it makes sense)
    - Determine stride step for the kernel (filter) at each step
  - To increase performance, consider holding on pooling for several steps:

    activation > conv2d > activation > conv2d > max_pool

    This way you use more intermediate features initially to maximize the effect of training at the cost of computational efficiency

Model

- The architecture of this model is simple yet it performs well for illustrating different techniques used in image classification and recognition

```
# Converting the images to a float of [0,1) to match the ex-
pected input to convolution2d
float_image_batch = tf.image.convert_image_dtype(image_batch,
tf.float32)

conv2d_layer_one = tf.contrib.layers.convolution2d(
    float_image_batch,
    num_output_channels=32,     # The number of filters to gen-
erate
    kernel_size=(5,5),          # It's only the filter height
and width.
```

Model

```
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(2, 2),
    trainable=True)
pool_layer_one = tf.nn.max_pool(conv2d_layer_one,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')

# Note, the first and last dimension of the convolution output
hasn't changed but the
# middle two dimensions have.
conv2d_layer_one.get_shape(), pool_layer_one.get_shape()
```

- The output from executing the example code is:

```
 (TensorShape([Dimension(3), Dimension(125), Dimension(76), Di-
mension(32)]),
    TensorShape([Dimension(3), Dimension(63), Dimension(38),
Dimension(32)]))
```

Model

- After a convolution is applied to the images, the output is downsized using a max_pool operation
- After the operation, the output shape of the convolution is reduced in half due to the ksize used in the pooling and the strides
- The reduction didn't change the number of filters (output channels) or the size of the image batch
- The components that were reduced dealt with the height and width of the image (filter)

Model

```
conv2d_layer_two = tf.contrib.layers.convolution2d(
    pool_layer_one,
    num_output_channels=64,         # More output channels
means an increase in the number of filters
    kernel_size=(5,5),
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(1, 1),
    trainable=True)

pool_layer_two = tf.nn.max_pool(conv2d_layer_two,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')

conv2d_layer_two.get_shape(), pool_layer_two.get_shape()
```

- The output from executing the example code is:

```
 (TensorShape([Dimension(3), Dimension(63), Dimension(38), Di-
mension(64)]),
    TensorShape([Dimension(3), Dimension(32), Dimension(19),
Dimension(64)]))
```

Model

- The tensor being operated on is still fairly complex tensor, the next step is to fully connect every point in each image with an output neuron

- Since this example is using softmax later, the fully connected layer needs to be changed into a rank two tensor

- The tensor's 1st dimension will be used to separate each image while the 2nd dimension is a rank one tensor of each input tensor

Model

```
flattened_layer_two = tf.reshape(
    pool_layer_two,
    [
        batch_size,  # Each image in the image_batch
        -1           # Every other dimension of the input
    ])

flattened_layer_two.get_shape()
```

- The output from executing the example code is:

```
TensorShape([Dimension(3), Dimension(38912)])
```

- tf.reshape has a special value that can be used to flatten all the remaining dimensions

Model

```
# The weight_init parameter can also accept a callable, a lamb-
da is used here  returning a truncated normal
# with a stddev specified.
hidden_layer_three = tf.contrib.layers.fully_connected(
    flattened_layer_two,
    512,
    weight_init=lambda i, dtype: tf.truncated_normal([38912,
512], stddev=0.1),
    activation_fn=tf.nn.relu
)

# Dropout some of the neurons, reducing their importance in
the model
hidden_layer_three = tf.nn.dropout(hidden_layer_three, 0.1)

# The output of this are all the connections between the previ-
ous layers and the 120 different dog breeds
# available to train on.
final_fully_connected = tf.contrib.layers.fully_connected(
    hidden_layer_three,
    120,  # Number of dog breeds in the ImageNet Dogs dataset
    weight_init=lambda i, dtype: tf.truncated_normal([512,
120], stddev=0.1)
)
```

Training

- Once a model is ready to be trained, the last steps follow the same process as discussed earlier
- The model's:
  - loss is computed based on how accurately it guessed the correct labels in the training data which feeds into a
  - training optimizer which updates the
  - weights of each layer
- This process continues one iteration at a time while attempting to increase the accuracy of each step

Training

- An important note related to this model, during training most classification functions (tf.nn.softmax) require numerical labels (as highlighted in the section describing loading the images from TFRecords)
- At this point, each label is a string similar to:
  n02085620-Chihuahua
- Instead of using tf.nn.softmax on this string, the label needs to be converted to be a unique number for each label
- Converting these labels into an integer representation should be done in preprocessing

Training

```
import glob

# Find every directory name in the imagenet-dogs directory
(n02085620-Chihuahua, ...)
labels = list(map(lambda c: c.split("/")[-1], glob.glob("./
imagenet-dogs/*")))

# Match every label from label_batch and return the index
where they exist in the list of classes
train_labels = tf.map_fn(lambda l: tf.where(tf.equal(labels,
l))[0,0:1][0], label_batch, dtype=tf.int64)
```

- This example code uses two different forms of a map operation

Training

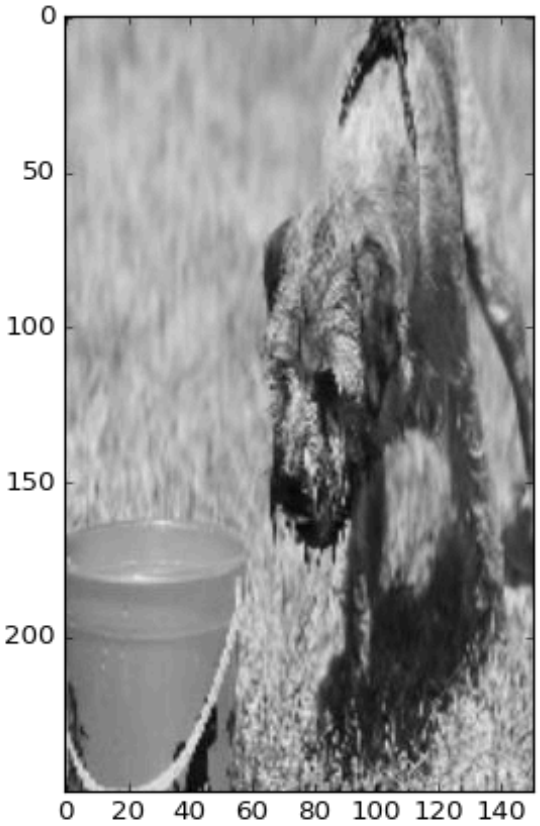
- The first form of map is used to create a list including only the dog breed name based on a list of directories
- The second form of map is tf.map_fn which is a TensorFlow operation that will map a function over a tensor on the graph
- The tf.map_fn is used to generate a rank one tensor including only the integer indexes where each label is located in the list of all the class labels

Debug the Filters with Tensorboard

- CNNs have multiple moving parts which can cause issues during training resulting in poor accuracy
- Debugging problems in a CNN often starts with investigating how the filters (kernels) are changing every iteration
- Each weight used in a filter is constantly changing

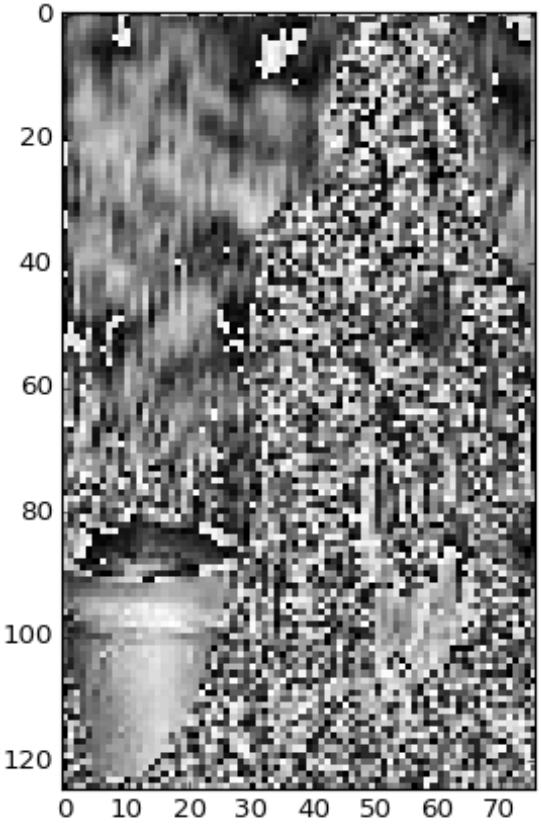
Debug the Filters with Tensorboard

- Here is an original grayscale training image before it is passed through the first convolution layer:



Debug the Filters with Tensorboard

- And, here is a single feature map from the first convolution layer highlighting randomness in the output:



CNN Conclusion

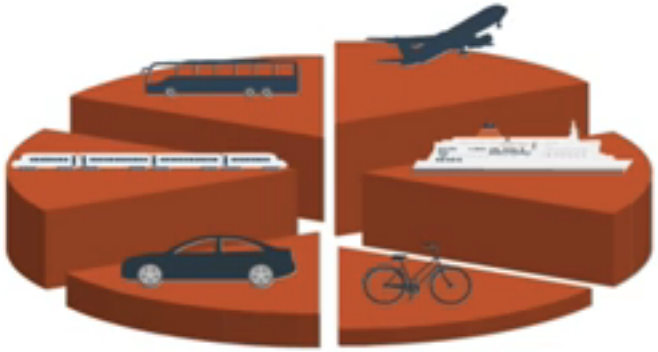
- CNN are a useful network architecture that are implemented with a minimal amount of code in TensorFlow
- While they're designed with images in mind, a CNN is not limited to image input
- Convolutions are used in multiple industries from music to medical and a CNN can be applied in a similar manner
- Currently, TensorFlow is designed for two dimensional convolutions but it's still possible to work with higher dimensionality input using TensorFlow

CNN Conclusion

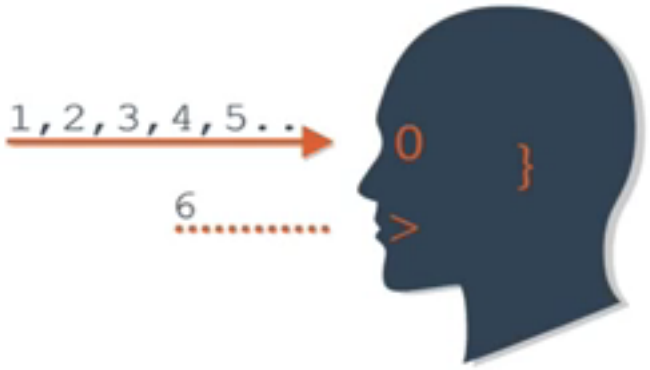
- While a CNN could theoretically work with natural language data (text), it isn't designed for this type of input
- Text input is often stored in a SparseTensor where the majority of the input is 0
- CNNs are designed to work with dense input where each value is important and the majority of the input is not 0
- Working with text data is a challenge which is addressed with the use of Recurrent Neural Networks (RNN) and Natural Language Processing (NLP)

FFNN vs RNN

Feedforward net = Classifier/Regressor



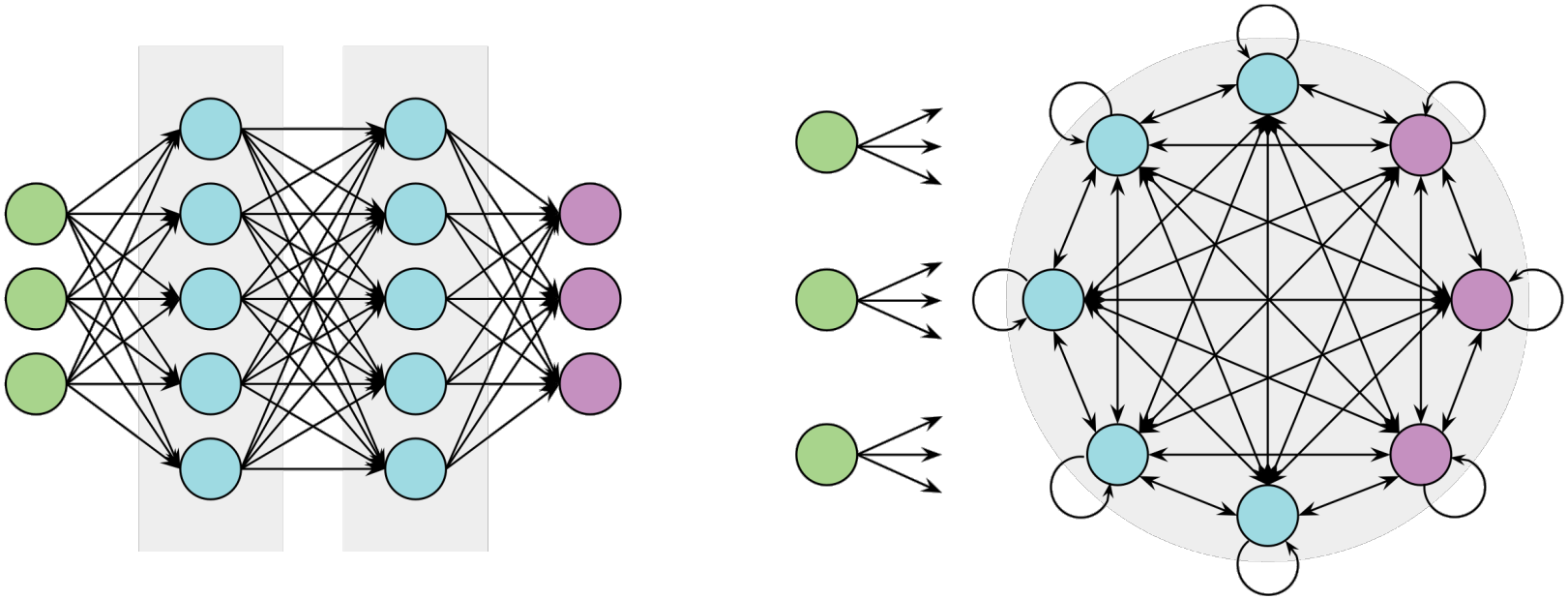
Recurrent Net = Forecaster



Recurrent Neural Networks and Natural Language Processing

- Recurrent Neural Networks (RNN) are a family of networks that explicitly model time
- RNNs build on the same neurons summing up weighted inputs from other neurons
- However, neurons are allowed to connect both forward to higher layers and backward to lower layers and form cycles
- The hidden activations of the network are remembered between inputs of the same sequence.

Recurrent Neural Networks and Natural Language Processing



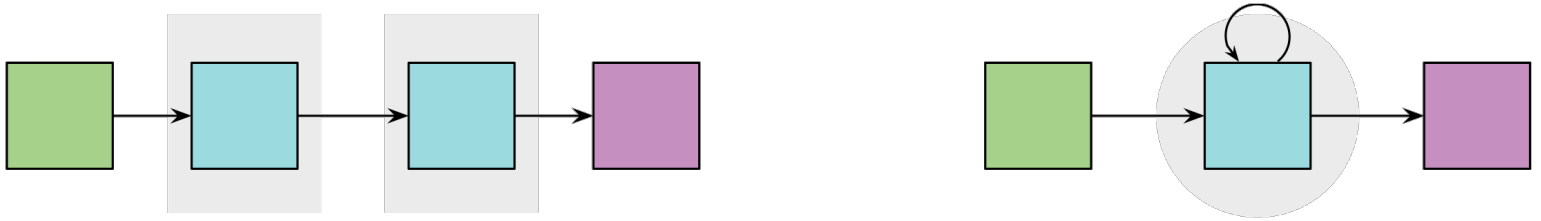
Feed Forward Network and Recurrent Network

Approximating Arbitrary Programs

- RNNs are arbitrary graphs of neurons and weights
- Input neurons have incoming connections because their activation is set by the input data
- The output neurons are just set of neurons in the graph that we read the prediction from
- All other neurons in the graph are referred to as hidden neurons
- The current hidden activations of an RNN are called state

Approximating Arbitrary Programs

- At the beginning of each sequence, we usually start with an empty state, initialized to zeros:

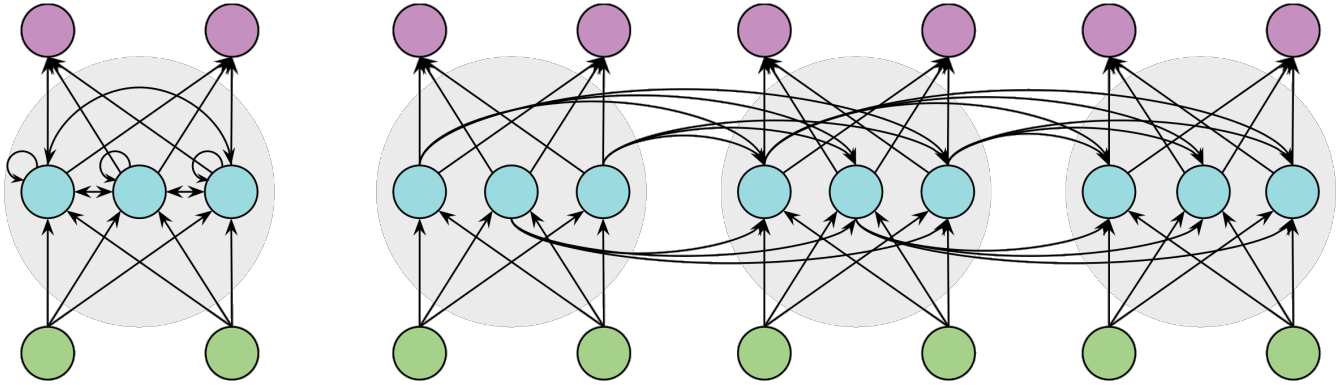


Short Notation of FFNN and RNN

- The state of an RNN depends on the current input and the previous state, which in turn depends on the input and state before that
- Therefore, the state has indirect access to all previous inputs of the sequence and can be interpreted as a working memory

Backpropagation Through Time

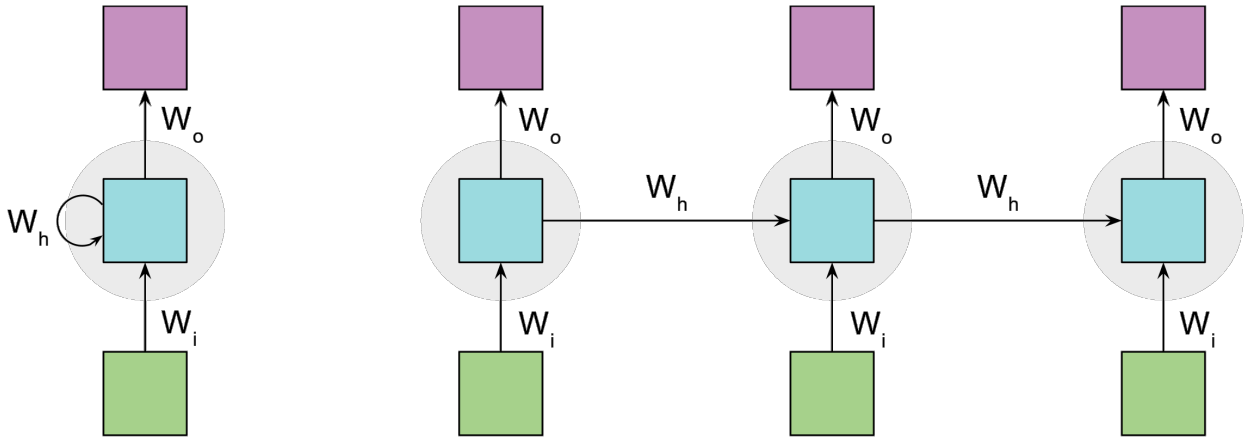
- As with forward networks, the most popular optimization method is based on Gradient Descent
- However, it is not straight-forward how to backpropagate the error in this dynamic system



Unfolding a Recurrent Network in Time

Backpropagation Through Time

- We can apply standard backpropagation through this unfolded RNN in order to compute the gradient of the error with respect to the weights

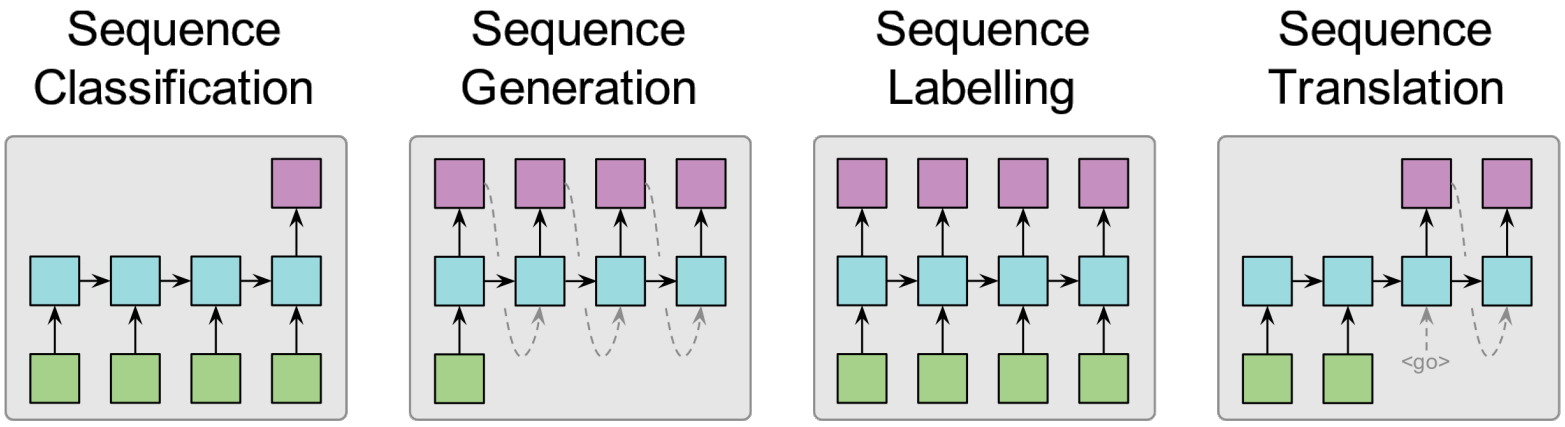


Unfolding an RNN in Time in Short Notation

- This algorithm is called Back-Propagation Through Time

Encoding and Decoding Sequences

- Sometimes, the input is a sequence and the output is a single vector, or the other way around
- RNNs can handle those and more complicated cases well



Common Mappings with Recurrent Networks

Implementing a Recurrent Network

- TensorFlow supports various variants of RNNs that can be found in the tf.nn.rnn_cell module
- With the tf.nn.dynamic_rnn() operation, TensorFlow also implements the RNN dynamics for us
- We therefore use the newer dynamic_rnn() operation

Implementing a Recurrent Network

```
import tensorflow as tf

# The input data has dimensions batch_size * sequence_length *
frame_size.
# To not restrict ourselves to a fixed batch size, we use None
as size of
# the first dimension.
sequence_length = ...
frame_size = ...
data = tf.placeholder(tf.float32, [None, sequence_length,
frame_size])

num_neurons = 200
network = tf.nn.rnn_cell.BasicRNNCell(num_neurons)

# Define the operations to simulate the RNN for se-
quence_length steps.
outputs, states = tf.nn.dynamic_rnn(network, data,
dtype=tf.float32)
```

A quick work on Optimizers

- Which one to chose?
  - In a NN we want to converge faster, learn properly and tune the hyperparameters in order to minimize the loss function
- Types of Optimizers:
  - **1. Gradient Descent**:
    - is the most important technique used for training and optimization in NNs
    - It will find the Minima, then control the variance, then update the Model's parameters, and finally will lead to Convergence
  - **2. Stochastic Gradient Descent**:
    - it performs a parameter update for each training example
    - It performs one update at a time, but it is faster
    - Frequent updates cause high variance and causes the loss function to fluctuate
    - This is good because we can discover better Minima (as oppose to GD)
    - However, this can complicate convergence to an exact minimum

A quick work on Optimizers

- Which one to chose?
  - In a NN we want to converge faster, learn properly and tune the hyperparameters in order to minimize the loss function
- Types of Optimizers:
  - **3. Adam** (adaptive learning):
    - very frequently used because it outperforms other adaptive techniques
    - Provides: fast convergence, it works well in complex NNs,
  - **4. Adagrad** (adaptive learning):
    - it makes big updates for infrequent parameters and small updates for frequent parameters
    - Good for sparse data
  - For sparse datasets it is always better to use adaptive learning rate techniques !

A quick work on Optimizers

- Which one to chose?
  - In a NN we want to converge faster, learn properly and tune the hyperparameters in order to minimize the loss function
- Types of Optimizers:

TensorFlow

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

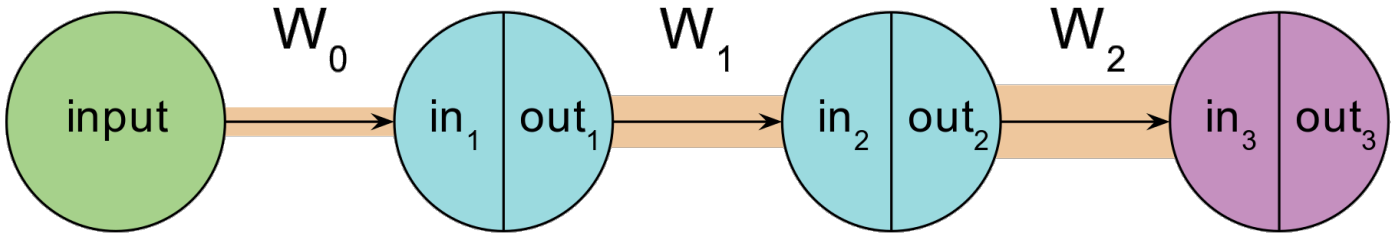
`class Adamax` : Optimizer that implements the Adamax algorithm.

Vanishing and Exploding Gradients

- The reason why it is difficult for an RNN to learn such long-term dependencies lies in how errors are propagated trough the network during optimization
- For long sequences, the unfolded network gets very deep and has many layers
- At each layer, backpropagation scales the error from above in the network by the local derivatives

Vanishing and Exploding Gradients

- If most of local derivatives are much smaller than the value of one, the gradient gets scaled down at every layer causing it to shrink exponentially and eventually, vanish
- Analogously, many local derivatives being greater than one cause the gradient to explode



Unstable Gradients in Deep Networks

Vanishing and Exploding Gradients

- Let's compute the gradient of this example network with just one hidden neuron per layer in order to get a better understanding of this problem
- For each layer *i* the local derivatives $f'(in_i) * W_i^T$ get multiplied together:

$$\frac{\partial C}{\partial in_1} = \frac{\partial C}{\partial out_3}\frac{\partial out_3}{\partial in_3} * \frac{\partial in_3}{\partial out_2}\frac{\partial out_2}{\partial in_2} * \frac{\partial in_2}{\partial out_1}\frac{\partial out_1}{\partial in_1}$$

- Resolving the derivatives yields:

$$cost'(f(out_3)) * f'(in_3) * \boxed{W_2^T} * f'(in_2) * \boxed{W_1^T} * f'(in_1)$$

Long-Short Term Memory

- LSTM is a special form of RNN that is designed to overcome the vanishing and exploding gradient problem
- The LSTM architecture replaces the normal neurons in an RNN with so-called LSTM cells that have a little memory inside
- Those cells are wired together as they are in a usual RNN but they have an internal state that helps to remember errors over many time steps

Long-Short Term Memory

- The trick of LSTM is that this internal state has a self-connection with a **fixed weight of one** and a linear activation function, so that its local derivative is always one

- During backpropagation, this so called constant error carousel can carry errors over many time steps without having the gradient vanish or explode

$$e_t = f'(in_t) * w * e_{t+1} = 1.0$$

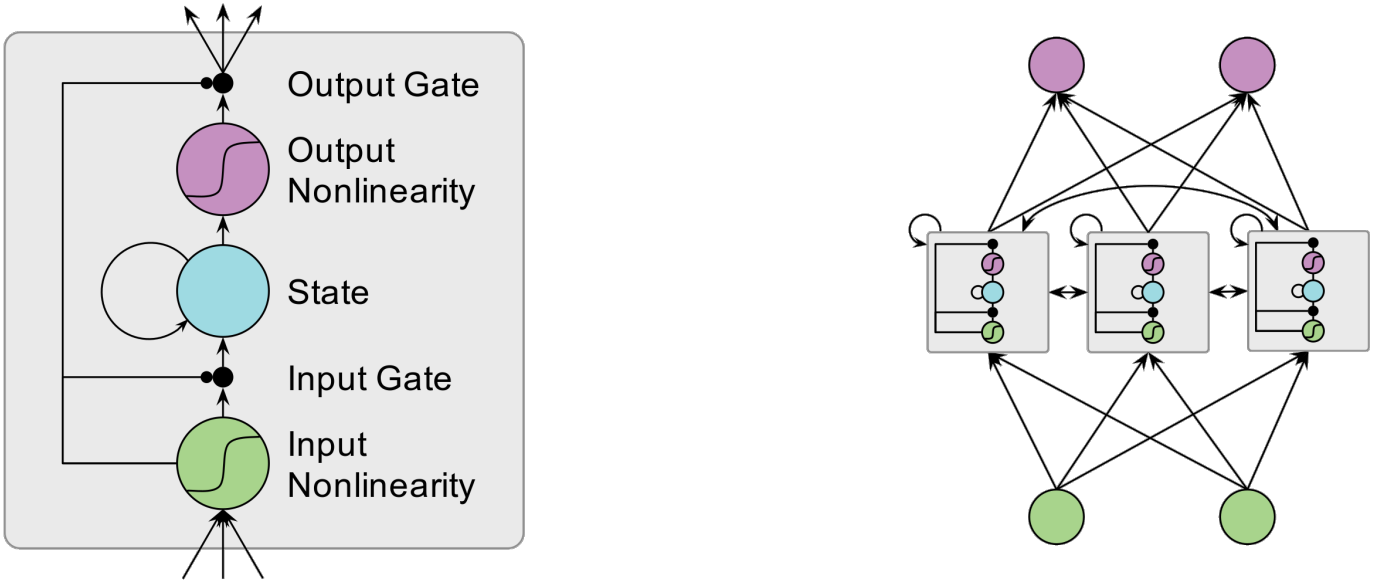
Long-Short Term Memory

- The purpose of the internal state is to deliver errors over many time steps, the LSTM architecture leaves learning to the surrounding gates that have non-linear, usually sigmoid, activation functions

- LSTM cell has two gates:
  - one learns to scale the incoming activation
  - one learns to scale the outgoing activation
- The cell can thus learn when to incorporate or ignore new inputs and when to release the feature it represents to other cells
- The input to a cell is fed into all gates using individual weights

Long-Short Term Memory

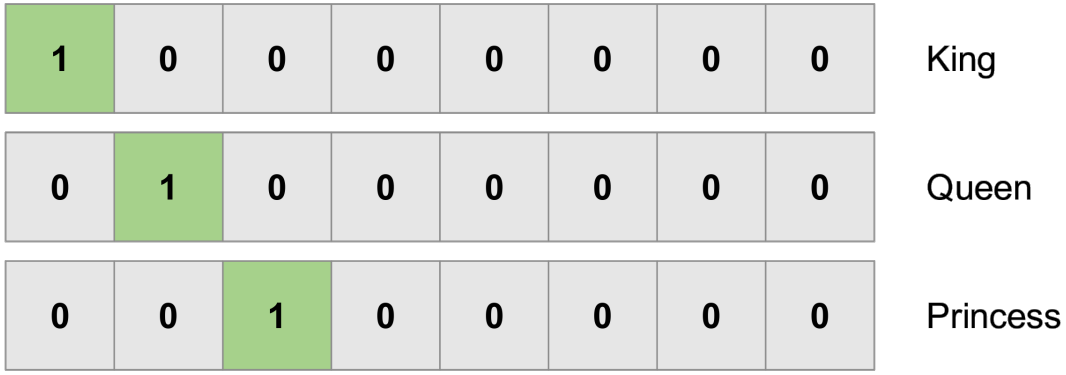
- TensorFlow provides an LSTM network via the LSTMCell class



Long Short-Term Memory (LSTM)

Word Vector Embedding

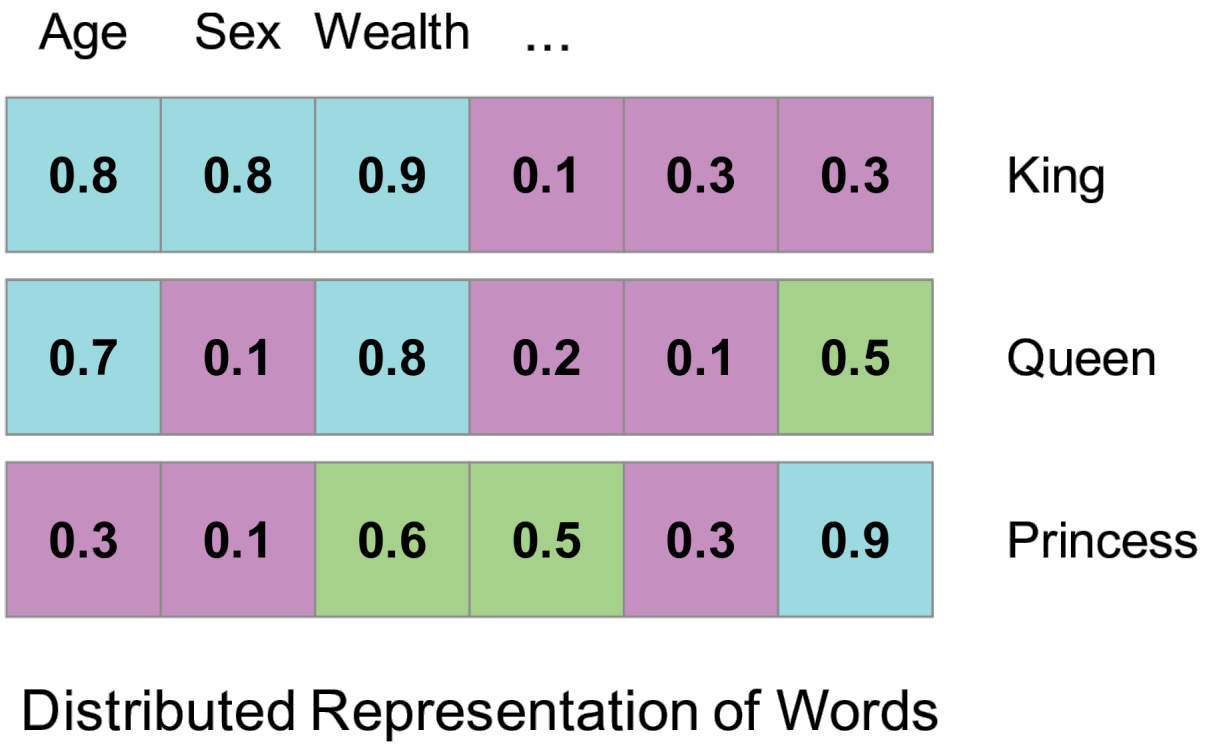
- Lets implement a model to learn word embedding, a very powerful way to represent words for NLP tasks
- Why to represent words as vectors?
  - The most straight-forward way to feed words into a learning system is one-hot encoded, that is, as a vector of the vocabulary size with all elements zero except for the position of that word set to one



One-Hot Encoded Representation of Words

Word Vector Embedding

- A solution to the semantic relatedness:



PCA

- Principal component analysis (PCA) is a statistical procedure that aims to convert a set of observations, possibly correlated, into a set of linearly uncorrelated variables called principal components

- PCA uses orthogonal transformation to achieve that goal

- The number of distinct principal components P_c is:

$$P_c = \min \{ \text{argmax}(Or_v[:]), \text{argmax}(Ob_v[:]) \} - 1$$

where,

Or_v is the number of the ***original variables*** / features / attributes

Ob_v is the number of ***observations***

PCA

- This transformation is defined so that:
  - the 1st P_c has the largest variance:
    - it accounts for the largest variability in the data
  - The 2nd P_c has the next highest variance while it is orthogonal to the preceding component (the 1st P_c),
  - and so on and so forth…
- The resulting vectors is an uncorrelated orthogonal set

PCA

- Principal Component Analysis (PCA) is a linear transformation algorithm

- PCA can also be described as a projection method as it projects observations from a *m-dimensional* space with *m* features to an *n-dimensional* space, and $n < m$ (hence reduction is achieved)

- In doing so, the maximum amount of information is conserved from the initial dimensions. (information = total variance of the dataset)

- If the information associated with the first few axes (PCA dimensions) represents a sufficient percentage of the total variability (on a scatter plot) => the observations could be represented on a simpler chart, thus making interpretation easier

PCA

- Principal Component Analysis (PCA) is a Data Mining method because it can easily extract information from large datasets

- Applications may be:
  - Visualization of the correlations between variables
  - Reducing the number of variables to be measured as a result
  - Obtaining non-correlated factors which are linear combinations of the initial variables to be used in modeling methods such as linear regression, discriminant analysis, logistic regression
  - Visualizing observations in a 2- or 3-dimensional space to identify uniform or atypical groups of observations

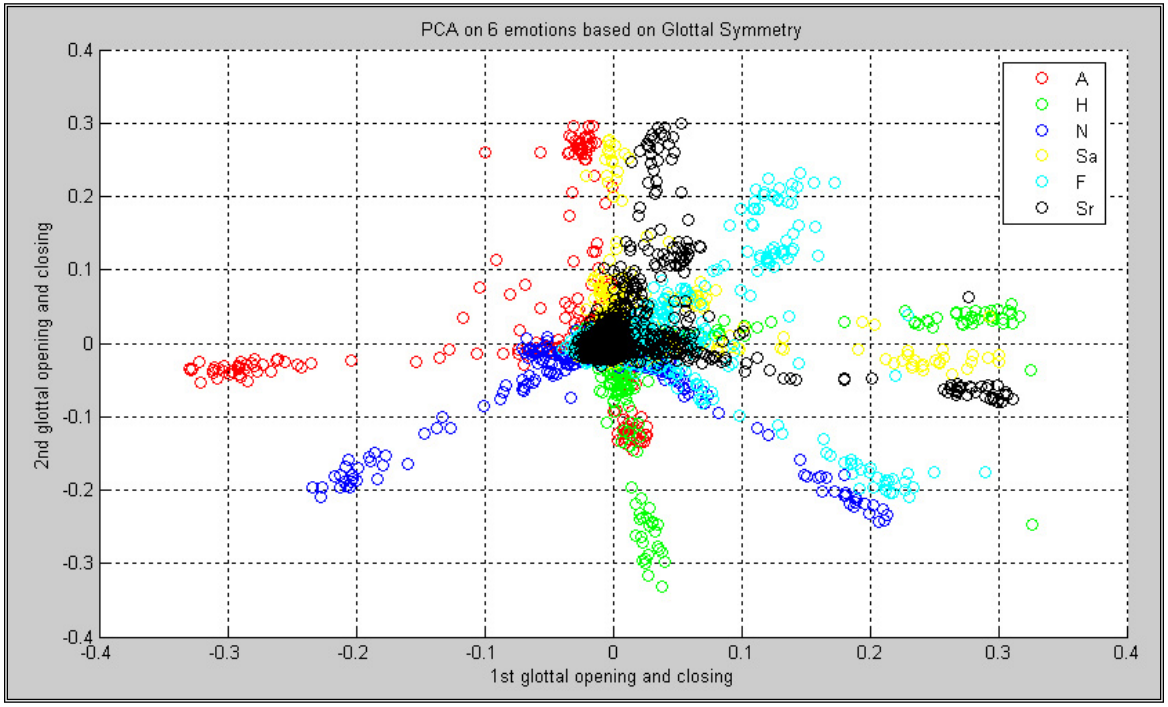
PCA

- Principal component analysis (PCA)
- If you add up the variances along each axis and then transform the points into a different coordinate system you get the same total variance in both cases
- This is always true provided the coordinate systems are orthogonal
- Rules:
  - choose the second axis in the way that maximizes the variance along it
  - same goes for the next axis, and so on and so forth…

PCA

- Principal component analysis (PCA)
- Examples:
  - 2D: Place the first axis in the direction of greatest variance of the points to maximize the variance along that axis. The second axis is perpendicular to it (so there is no choice here)
  - 3D: the second axis can lie anywhere in the plane so long it remains perpendicular to the first axis (so there is more choice here)
  - nD: in higher dimensions there is even more choice, though it is always constrained to be perpendicular to the first axis

PCA



PCA analysis of the 1st vs. 2nd Glottal Symmetry for 6 emotions

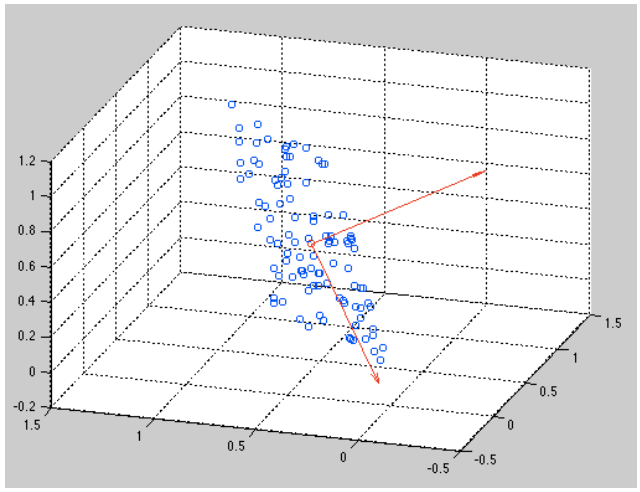
PCA

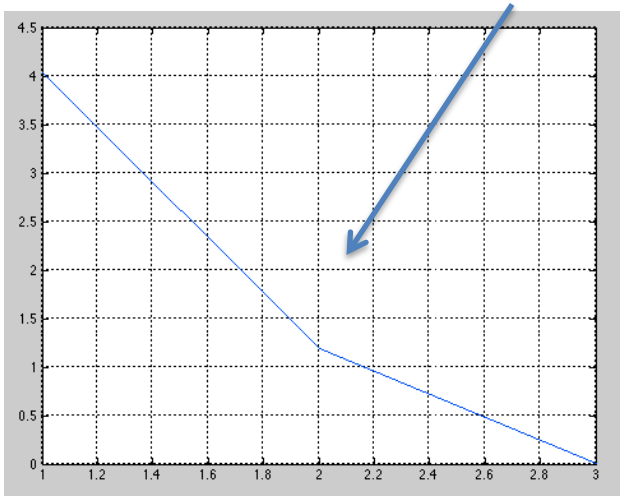
- PCA Example in Matlab:

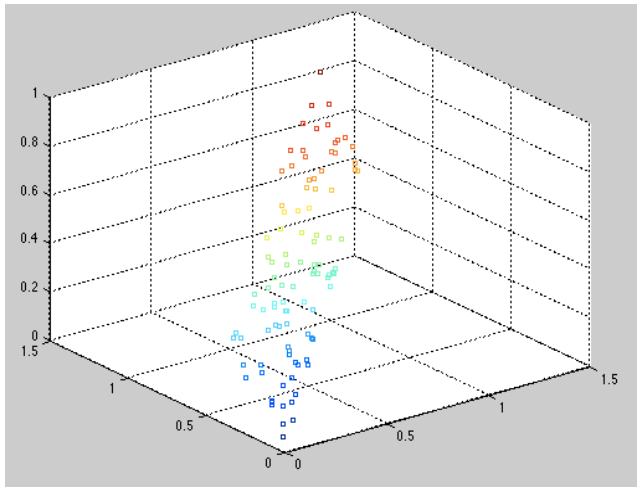
recall

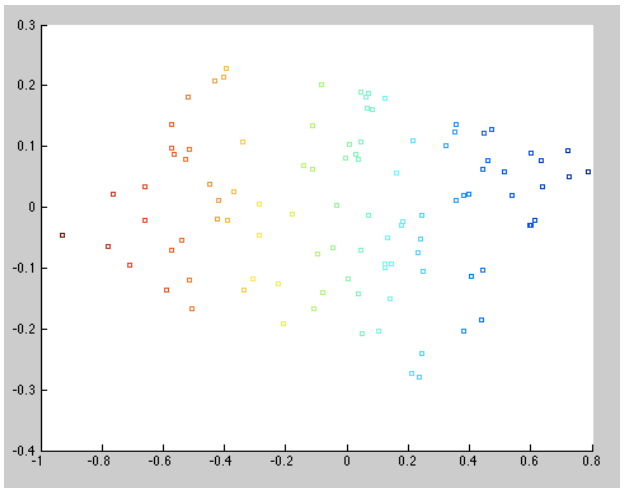
The number of distinct principal components P_c is:

$P_c = \min \{ \text{argmax}(Or_v[:]), \text{argmax}(Ob_v[:]) \} - 1$







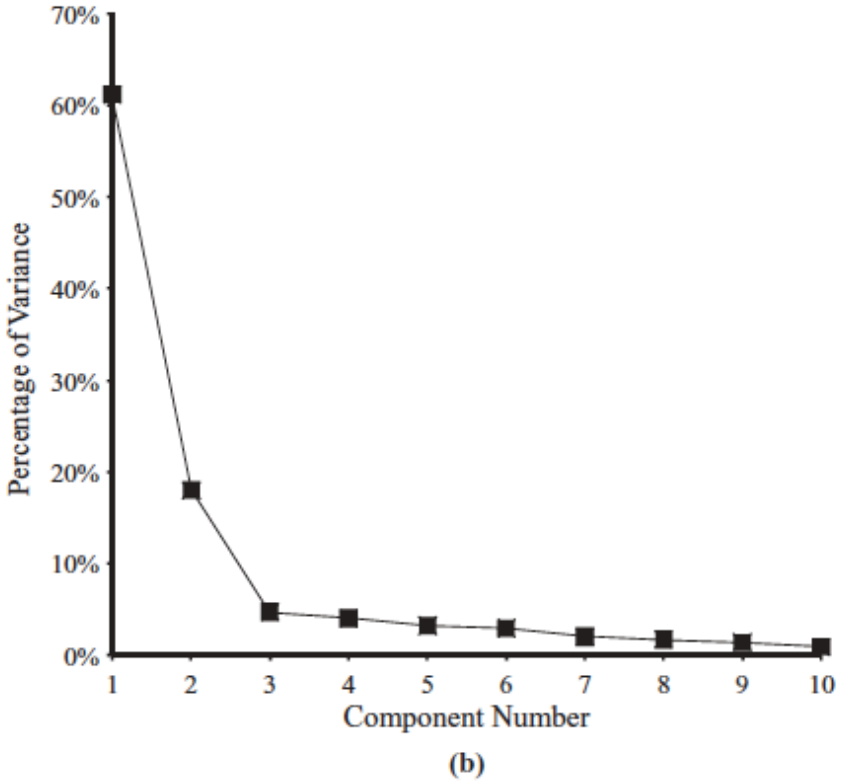


PCA

- PCA Example:

Axis	Variance	Cumulative
1	61.2%	61.2%
2	18.0%	79.2%
3	4.7%	83.9%
4	4.0%	87.9%
5	3.2%	91.1%
6	2.9%	94.0%
7	2.0%	96.0%
8	1.7%	97.7%
9	1.4%	99.1%
10	0.9%	100.0%

(a)



(b)

Principal components transform of a dataset: (a) variance of each component and (b) variance plot.