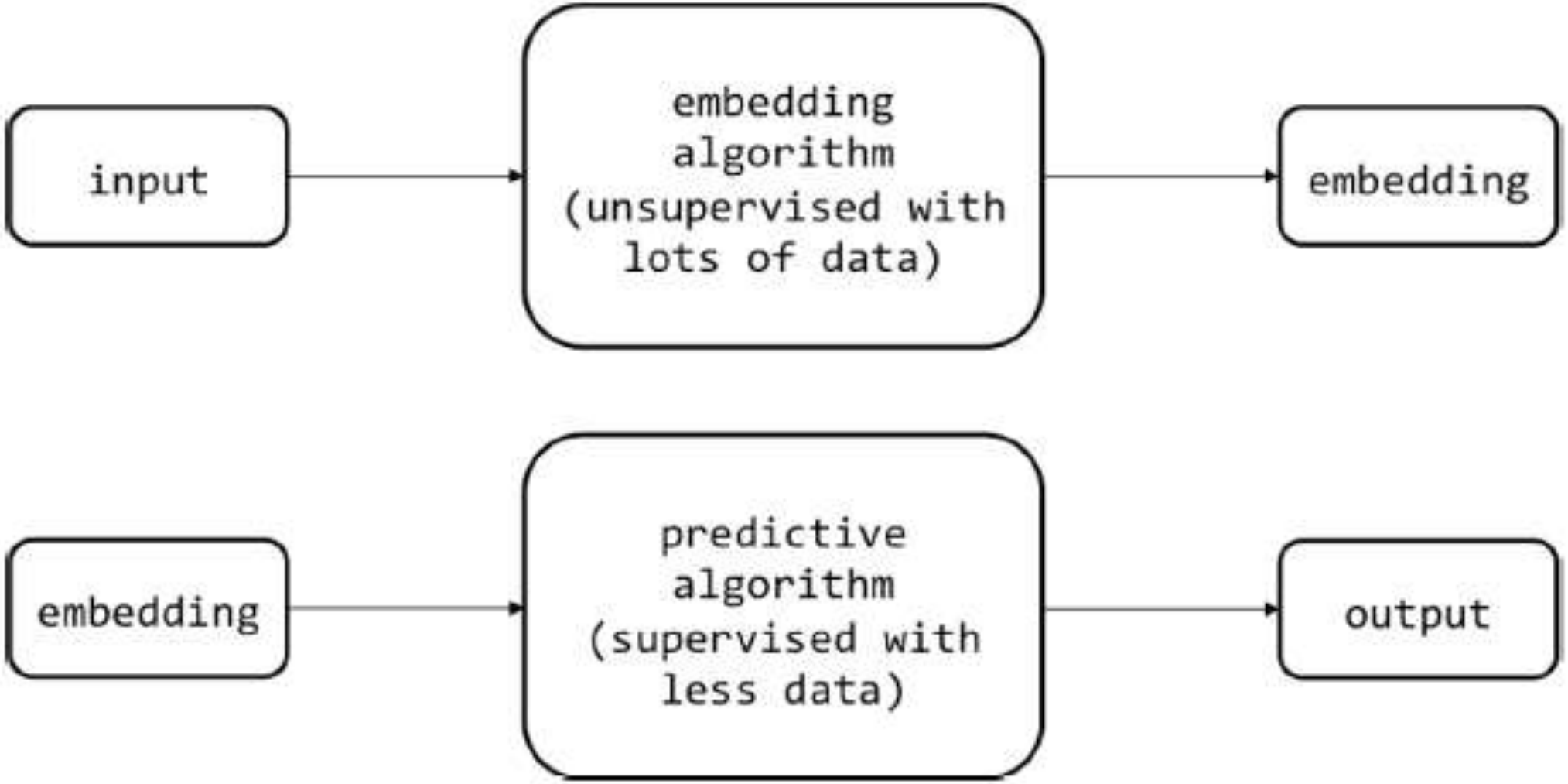
Embedding and Representation Learning

Dr. Sanjay Chatterji

CSE733

Learning Lower-Dimensional Representation

- Convolutional architecture uses simple argument.
  - The larger our input vector, the larger our model.
  - It is expressive, but increasingly data hungry.
  - Without sufficiently large volumes of training data, it will likely overfit.
  - It helps us cope with the curse of dimensionality by reducing the number of parameters without necessarily diminishing expressiveness.
  - still require large amounts of labeled training data.
- For many problems, labeled data is scarce and expensive to generate.
  - unlabeled data is plentiful.
- Our goal here is to develop effective learning models for such situation.
  - *Embeddings*
  - low-dimensional representations
  - Unsupervised fashion
  - Use generated embeddings to solve learning problems using smaller models.



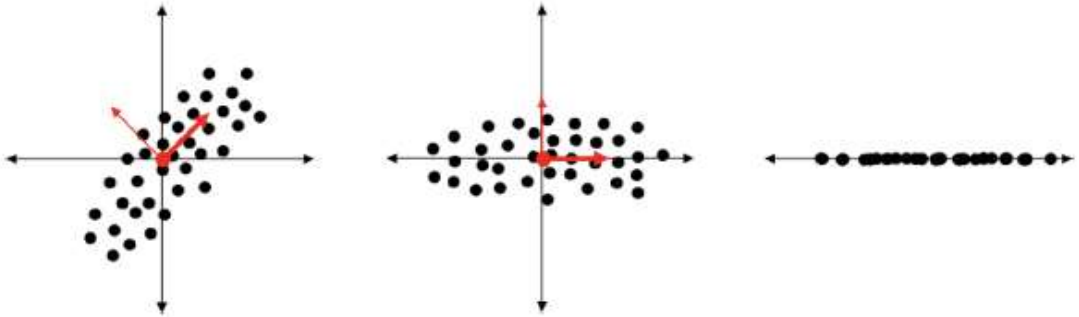
Using embeddings to automate feature selection in the case of scarce labeled data

To learn good embeddings

- We'll explore other applications of learning lower-dimensional representations, such as visualization and semantic hashing.
- We'll start by considering situations where all of the important information is already contained within the original input vector itself.
- In this case, learning embeddings is equivalent to developing an effective compression algorithm.
- We'll introduce principal component analysis (PCA), a classic method for dimensionality reduction.
- Then we'll explore more powerful neural methods for learning compressive embeddings.

Principal Component Analysis (PCA)

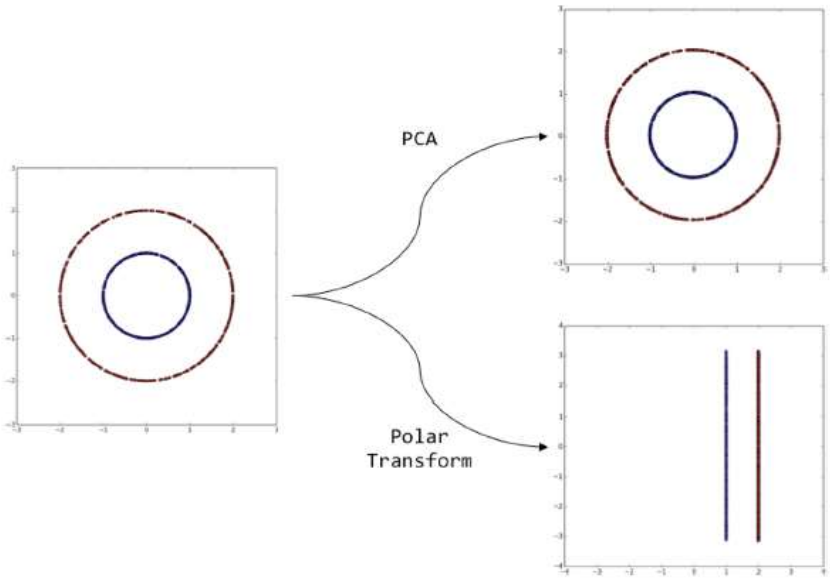
- A classic method for dimensionality reduction.
- If we have *d* dimensional data, we'd like to find a new set of $m < d$ dimensions that conserves as much valuable information from the original dataset.
- Assuming that variance corresponds to information, we can perform this transformation through an iterative process.
  - First Axis: unit vector along which the dataset has maximum variance.
  - Second Axis: From the set of vectors orthogonal to the first axis, we pick a new unit vector along which the dataset has maximum variance.
  - We continue this process until we have found a total of *d* new vectors in an axes.
  - We project our data onto this new set of axes.
- Let's choose $d = 2$, $m = 1$.



Mathematical Details

- We can view this operation as a project onto the vector space spanned by the top *m* eigenvectors of the dataset's covariance matrix.
- Let us represent the dataset as a matrix **X** with dimensions $n \times d$
- We'd like to create an embedding matrix **T** with dimensions $n \times m$.
- Using W where each column of **W** corresponds to an eigenvector of the matrix $\mathbf{X}^T\mathbf{X}$.
- It spectacularly fails to capture important relationships that are piecewise linear or nonlinear.

- We hope that PCA will transform
  - this two concentric circles
  - to a single new axis
  - that allows us to easily separate the red and blue dots.
  - information is being encoded in a nonlinear way
  - polar transformation: points as distance from the origin
- We need a theory for nonlinear dimensionality reduction
  - Deep learning practitioners have closed this gap

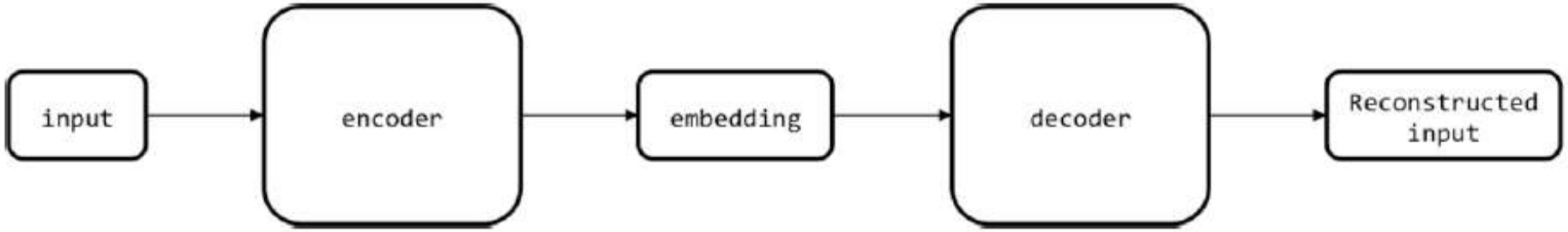


Motivating the Autoencoder Architecture

- We have discussed how each layer (in FFD) learned progressively more relevant representations of the input.
- We took the output of the final convolutional layer and used that as a lower-dimensional representation of the input image.
- The final output is a lower-dimensional representation of the input image.
- Problems with these approaches
  - while the selected layer contains information from the input, the network has been trained to pay attention to the aspects of the input that are critical to solving the task at hand.
  - there's a significant amount of information loss that may be important for other tasks, but potentially less important than the one immediately at hand

Motivating the Autoencoder Architecture

- *Autoencoder: encoder-decoder*
  - We first take the input and compress it into a low-dimensional vector.
    - it is responsible for producing the low-dimensional embedding or code
  - Then invert the computation and reconstruct the original input.
    - instead of mapping the embedding to an arbitrary label as we would in a feed-forward network
  - Surprising effectiveness
- The autoencoder architecture attempts to construct a high-dimensional input into a low-dimensional embedding and then uses that low-dimensional embedding to reconstruct the input



Implementing an Autoencoder in TensorFlow

- “Reducing the dimensionality of data with neural networks” by Hinton and Salakhutdinov in 2006
  - nonlinear complexities afforded by a neural model would allow them to capture structure that linear methods, such as PCA, would miss.
  - they ran an experiment on MNIST using both an autoencoder and PCA
  - reduce the dataset into two-dimensional data points
- The two dimensional embedding is now treated as the input, and the network attempts to reconstruct the original image.
  - inverse operation
  - we architect the decoder network so that the autoencoder has the shape of an hourglass.
- The output of the decoder network is a 784-dimensional vector that can be reconstructed into a 28 × 28 image.

https://www.heywhale.com/mw/project/5bfb8bdc954d6e0010676433

```
def encoder(x, n_code, phase_train):
    with tf.variable_scope("encoder"):
        with tf.variable_scope("hidden-layer-1"):
            hidden_1 = layer(x, [n_input, n_hidden_1], [n_hidden_1], mode_train)
        with tf.variable_scope("hidden-layer-2"):
            hidden_2 = layer(hidden_1, [n_hidden_1, n_hidden_2], [n_hidden_2], mode_train)
        with tf.variable_scope("hidden-layer-3"):
            hidden_3 = layer(hidden_2, [n_hidden_2, n_hidden_3], [n_hidden_3], mode_train)
        with tf.variable_scope("code"):
            code = layer(hidden_3, [n_hidden_3, n_code], [n_code], mode_train)
    return code
```

Decoder and Layer

```
def decoder(code, n_code, phase_train):
        with tf.variable_scope("decoder"):
                with tf.variable_scope("hidden_1"):
                        hidden_1 = layer(code, [n_code, n_decoder_hidden_1],
                                                [n_decoder_hidden_1], phase_train)
                with tf.variable_scope("hidden_2"):
                        hidden_2 = layer(hidden_1, [n_decoder_hidden_1, n_decoder_hidden_2],
                                                [n_decoder_hidden_2], phase_train)
                with tf.variable_scope("hidden_3"):
                        hidden_3 = layer(hidden_2, [n_decoder_hidden_2, n_decoder_hidden_3],
                                                [n_decoder_hidden_3], phase_train)
                with tf.variable_scope("output"):
                        output = layer(hidden_3, [n_decoder_hidden_3, 784], [784], phase_train)
        return output

def layer(input, weight_shape, bias_shape, phase_train):
        weight_init = tf.random_normal_initializer(stddev=(1.0/weight_shape[0])**0.5)
        bias_init = tf.constant_initializer(value=0)
        W = tf.get_variable("W", weight_shape,  initializer=weight_init)
        b = tf.get_variable("b", bias_shape, initializer=bias_init)
        logits = tf.matmul(input, W) + b
        return tf.nn.sigmoid(layer_batch_norm(logits, weight_shape[1], phase_train))
```

Implementing an Autoencoder in TensorFlow

- In order to accelerate training, we'll reuse the batch normalization strategy.
- To visualize the results, avoid introducing sharp transitions in neurons.
- For that sigmoidal neurons are used instead of our usual ReLU neurons.
- Construct a measure(objective function) to see how well our model functions.
- L2 norm of the difference between the two vectors: $\| I - O \| = \sqrt{\Sigma i \ (Ii - Oi)^2}$
- We average this function over the whole minibatch to generate our final objective function.
- Finally, we'll train the network using the Adam optimizer, logging a scalar summary of the error incurred at every minibatch.
- In TensorFlow, we can concisely express the loss and training operations.

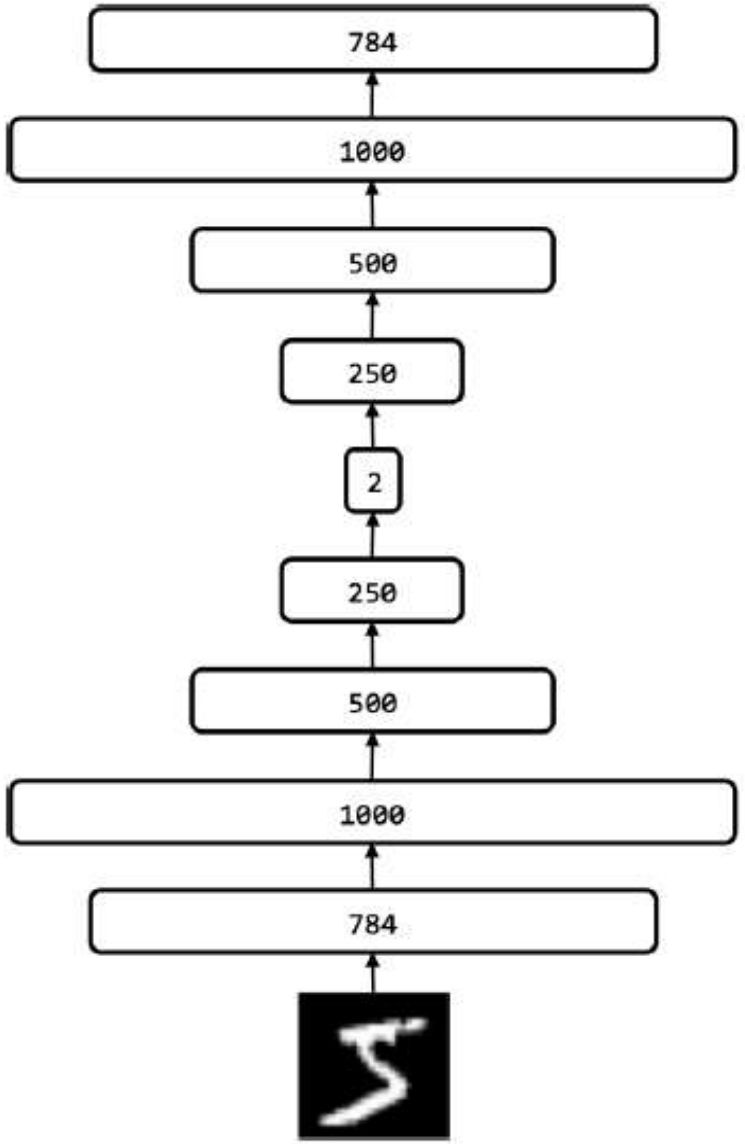
```
def loss(output, x):
        with tf.variable_scope("training"):
                l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x)), 1))
                train_loss = tf.reduce_mean(l2)
                train_summary_op = tf.scalar_summary("train_cost", train_loss)
                return train_loss, train_summary_op


def training(cost, global_step):
        optimizer = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08,
                                           use_locking=False, name='Adam')
        train_op = optimizer.minimize(cost, global_step=global_step)
        return train_op


def image_summary(summary_label, tensor):
        tensor_reshaped = tf.reshape(tensor, [-1, 28, 28, 1])
        return tf.image_summary(summary_label, tensor_reshaped)


def evaluate(output, x):
        with tf.variable_scope("validation"):
                in_im_op = image_summary("input_image", x)
                out_im_op = image_summary("output_image", output)
                l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x, name="val_diff")), 1))
                val_loss = tf.reduce_mean(l2)
                val_summary_op = tf.scalar_summary("val_cost", val_loss)
                return val_loss, in_im_op, out_im_op, val_summary_op
```

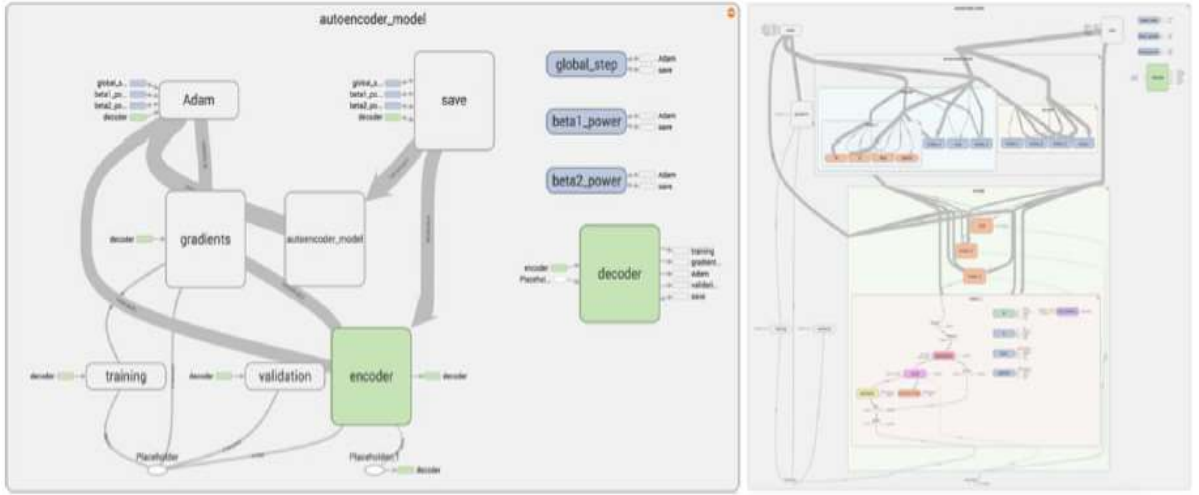
The experimental setup for dimensionality reduction of the MNIST dataset employed by Hinton and Salakhutdinov, 2006



Implementing an Autoencoder in TensorFlow

- Evaluate the generalizability of the model using validation dataset with L2 norm.
- We'll collect image summaries so that we can compare both the input images and the reconstructions.
- Finally, build the model out of these subcomponents and train the model.
- We accept command-line parameter for the number of neurons in our code layer.
  $python ae_mnist.py 2 will instantiate a model with two neurons in the code layer.
- We also reconfigure the model saver to maintain more snapshots of our model.
- We'll reload our most effective model to compare its performance to PCA later.
- We use summary writers to capture the image summaries after each epoch.

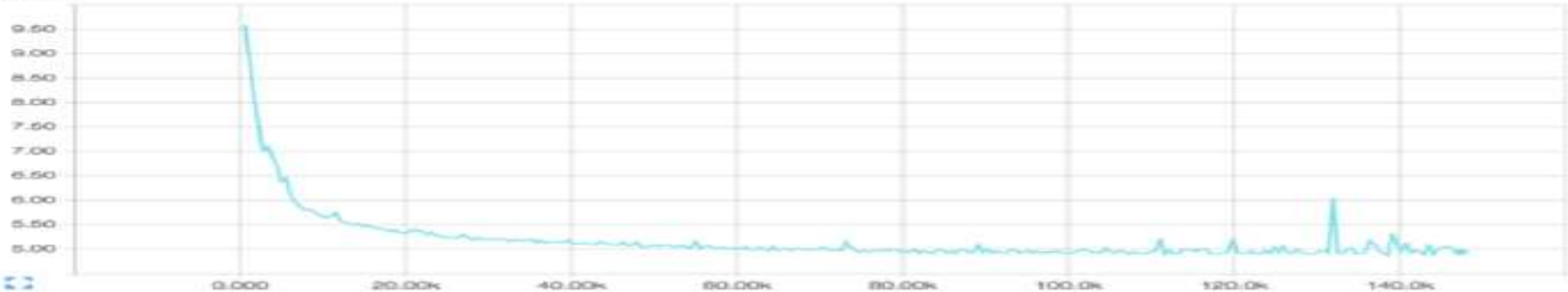
TensorBoard

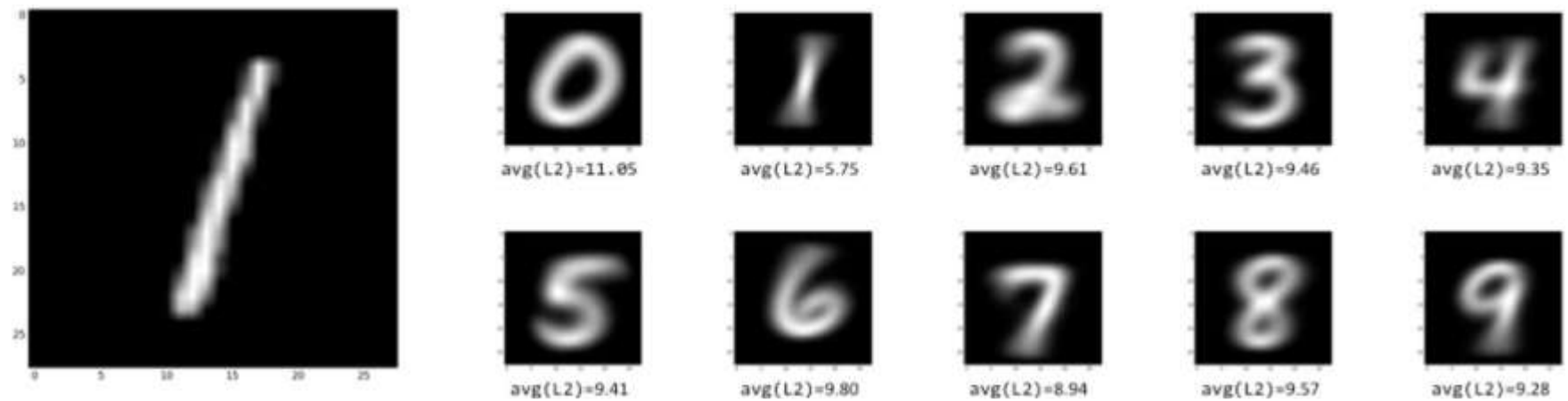


- We can visualize using TensorBoard
  - TensorFlow graph
  - training cost
  - validation costs
  - image summaries
- TensorBoard visualizations of the costs over the training and validation set.
- As we would expect for a successful model, both the training and validation curves decrease until they flatten off asymptotically. After approximately 200 epochs, we attain a validation cost of 4.78.
- *The image of the 1 on the left is compared to*
  - *all of the other digits in the MNIST dataset*
  - *for each digit class, we compute the average of the L2 costs, comparing X to each instance of the digit class.*
- *The reconstructions for three randomly chosen samples from the test set are shown.*

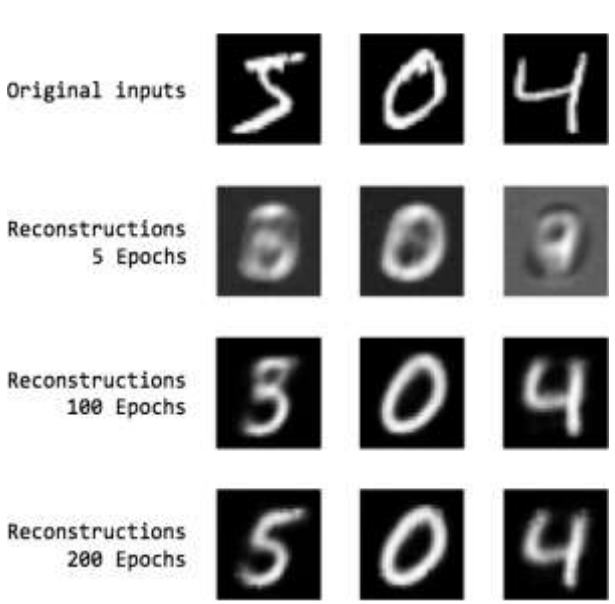
train_cost

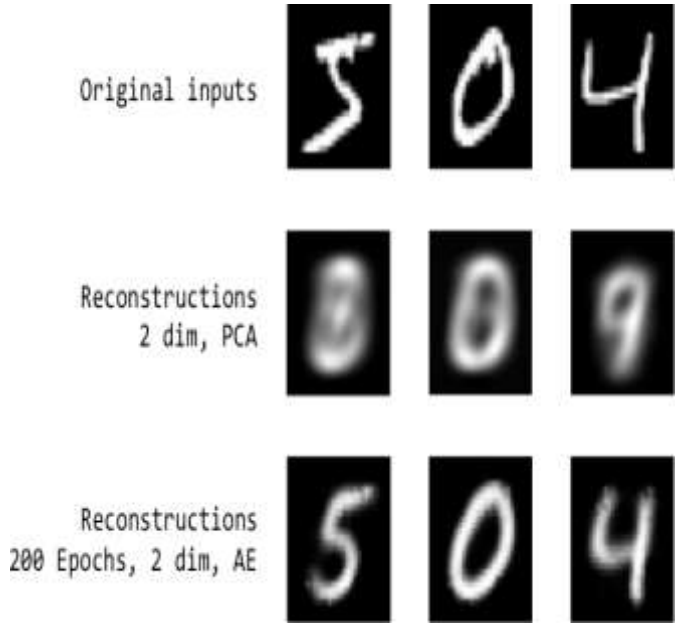
val_cost

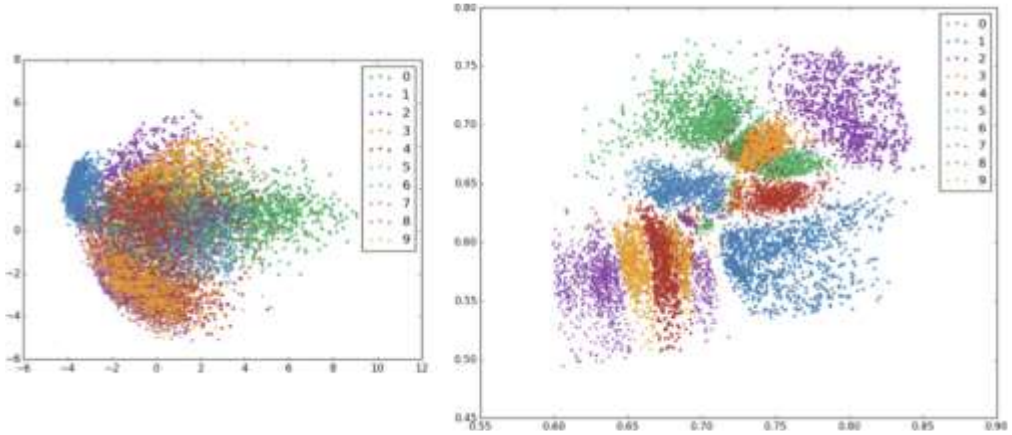




Reconstructions for three randomly chosen samples from the test set







Two dimensional embeddings produced by PCA (left) and by an autoencoder in clustering codes of different digit classes

2D codes produced by PCA and autoencoders

- We produce two-dimensional PCA codes on MNIST dataset.
  - from sklearn import decomposition
  - import input_data
  - mnist = input_data.read_data_sets("data/", one_hot=False)
  - pca = decomposition.PCA(n_components=2)
  - pca.fit(mnist.train.images)
  - pca_codes = pca.transform(mnist.test.images)

```
from matplotlib import pyplot as plt

pca_recon = pca.inverse_transform(pca_codes[:1])
plt.imshow(pca_recon[0].reshape((28,28)), cmap=plt.cm.gray)
plt.show()
```

- one-hot vector is of size 10 with the *ith* component set to one to represent digit *i* and the rest of the components set to zero
- PCA: It has trouble distinguishing 5's from 3's and 8's, 0's from 8's, and 4's from 9's.
- Repeating the same experiment with 30-dimensional codes provides significant improvement to the PCA reconstructions.
- A simple machine learning model more effectively classifies data points consisting of autoencoder embeddings as compared to PCA embeddings.

Denoising to Force Robust Representations

- *Denoising*: improve the ability of the autoencoder to generate embeddings that are resistant to noise.
- The human ability for perception is surprisingly resistant to noise.
- Even if we're exposed to a random sampling of pixels from an image, if we have enough information, our brain is still capable of concluding the ground truth of what the pixels represent with maximal probability.
- *Top row: original images from the MNIST dataset.*
- *Bottom row: randomly blacked out half of the pixels.*

Denoising Autoencoder

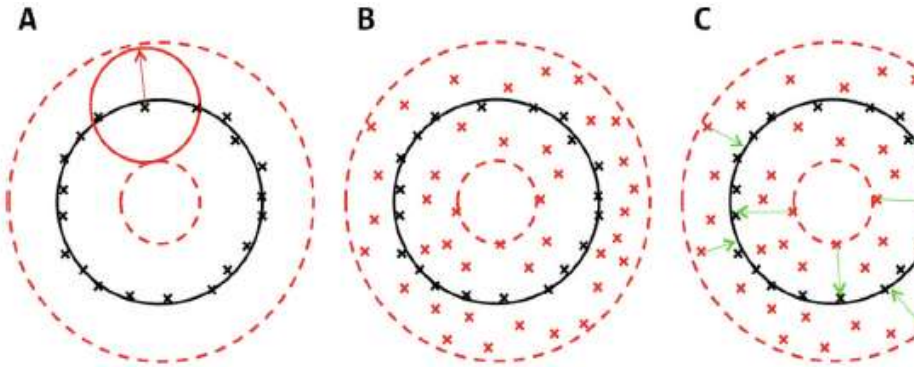
- Our mind is able to fill in the blanks to draw a conclusion.
- Even though only a corrupted version of a digit hits our retina, our brain is still able to reproduce the set of activations (i.e., the code or embedding) that we normally would use to represent the image of that digit.
- *Denoising Autoencoder (Vincent et al. in 2008)*
  - We corrupt a %age of pixels by setting them to zero.
  - Original image *X*, corrupted version *C(X)*
  - The denoising autoencoder is identical to the vanilla autoencoder except:
    - the input to this auto encoder network is the corrupted *C(X)* instead of *X*.
  - It learns a code for each input that is resistant to the corruption mechanism.
  - It is able to interpolate through the missing information to recreate the original image.

Geometric Explanation

- Let’s say we had a two dimensional dataset with various labels.
- S: All of the 2D data points in a particular category.
- Any arbitrary sampling of points could end up taking any form while visualized.
- Presume that for real-life categories, there is some underlying structure that unifies all of the points in *S*.
  - This underlying, unifying geometric structure is known as a *manifold*.
- We want to capture manifold when we reduce the dimensionality of our data.
- As it learns how to reconstruct data
  - The autoencoder is implicitly learning this manifold as it learns how to reconstruct data after pushing it through a bottleneck.
  - The autoencoder figures out a point belongs to one manifold or another as it tries to generate a reconstruction of an instance with potentially different labels.

Geometric Explanation

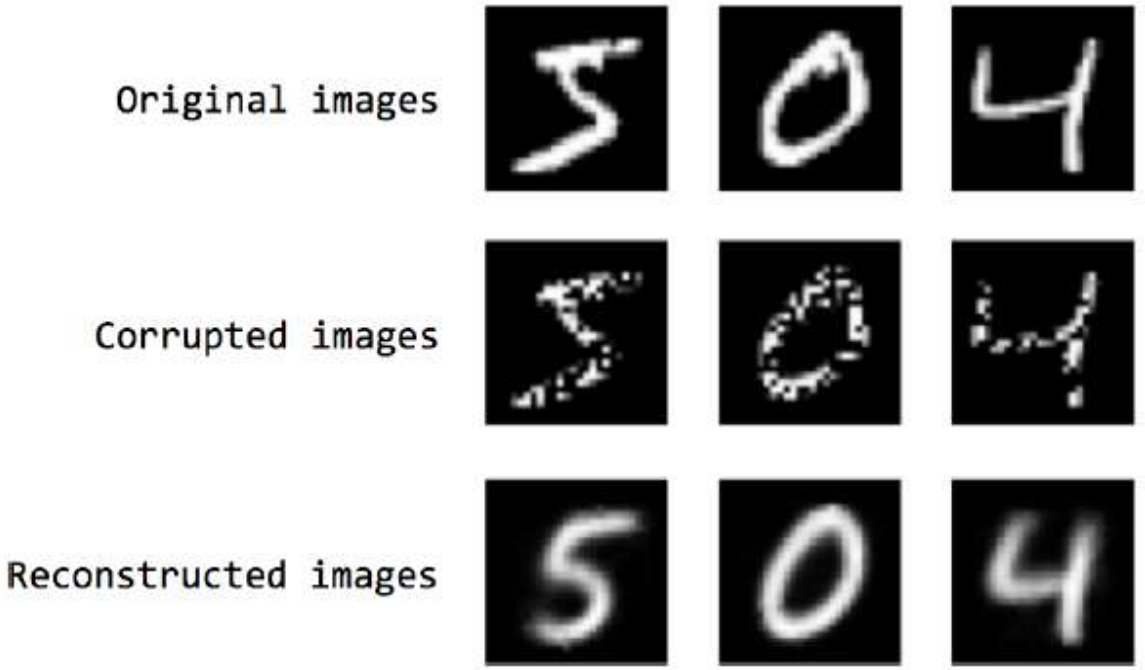
- Here the points in *S* are a simple low-dimensional manifold (Black Circle)
  - data points in *S* (black *'s)
  - the manifold that best describes them.
- Corruption operation
  - expands the dataset to not only include the manifold
  - but also all of the points in space around the manifold
  - up to a maximum margin of error.
- Denoising Autoencoder learns
  - to collapse all of the data points in this space back to the manifold.
  - which aspects of a data point are generalizable, broad strokes
  - which aspects are "noise"
  - to approximate the underlying manifold of *S*.



Changing autoencoder to build denoising autoencoder

- The code snippet corrupts the input if the corrupt placeholder is equal to 1
- It refrains from corrupting the input if the corrupt placeholder tensor = 0.
- After this modification, rerun the autoencoder, resulting the reconstructions.
- Denoising autoencoder has faithfully replicated our incredible human ability to fill in the missing pixels.

```
def corrupt_input(x):
    corrupting_matrix = tf.random_uniform(shape=tf.shape(x),
                                          minval=0,maxval=2,dtype=tf.int32)
    return x * tf.cast(corrupting_matrix, tf.float32)
    x = tf.placeholder("float", [None, 784]) # mnist data image of
    corrupt = tf.placeholder(tf.float32)
    phase_train = tf.placeholder(tf.bool)
    c_x = (corrupt_input(x) * corrupt) + (x * (1 - corrupt))
```

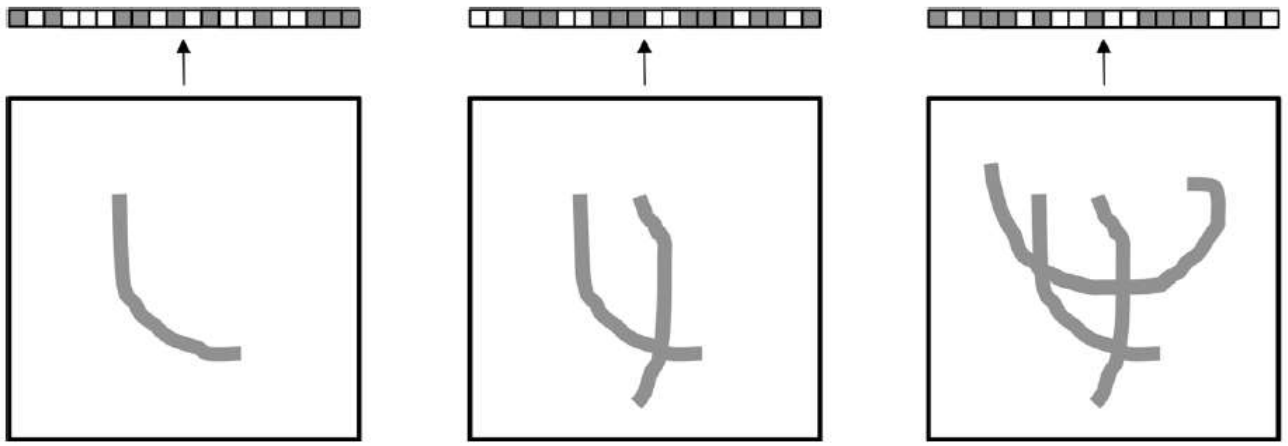


Interpretability

- Interpretability is a property of a machine learning model that measures how easy it is to inspect and explain its process and/or output.
- DNN
  - Interpretability is very less
  - due to nonlinearities and massive # of parameters.
  - more accurate
  - Less interpretability hinders their adoption in valuable but risky applications.
  - predicting that a patient has or does not have cancer
  - the doctor will likely want an explanation to confirm the model's conclusion
- We can address one aspect of interpretability by exploring the characteristics of the output of an autoencoder.
- Autoencoder's representations are dense, and we can see how the representation changes as we make modifications to the input.

Sparsity in Autoencoders

- The autoencoder produces a *dense* representation.
  - The representation of the original image is highly compressed.
- There are many dimensions to work with in the representation.
- The activations of a dense representation combine and overlay information from multiple features in ways that are difficult to interpret.

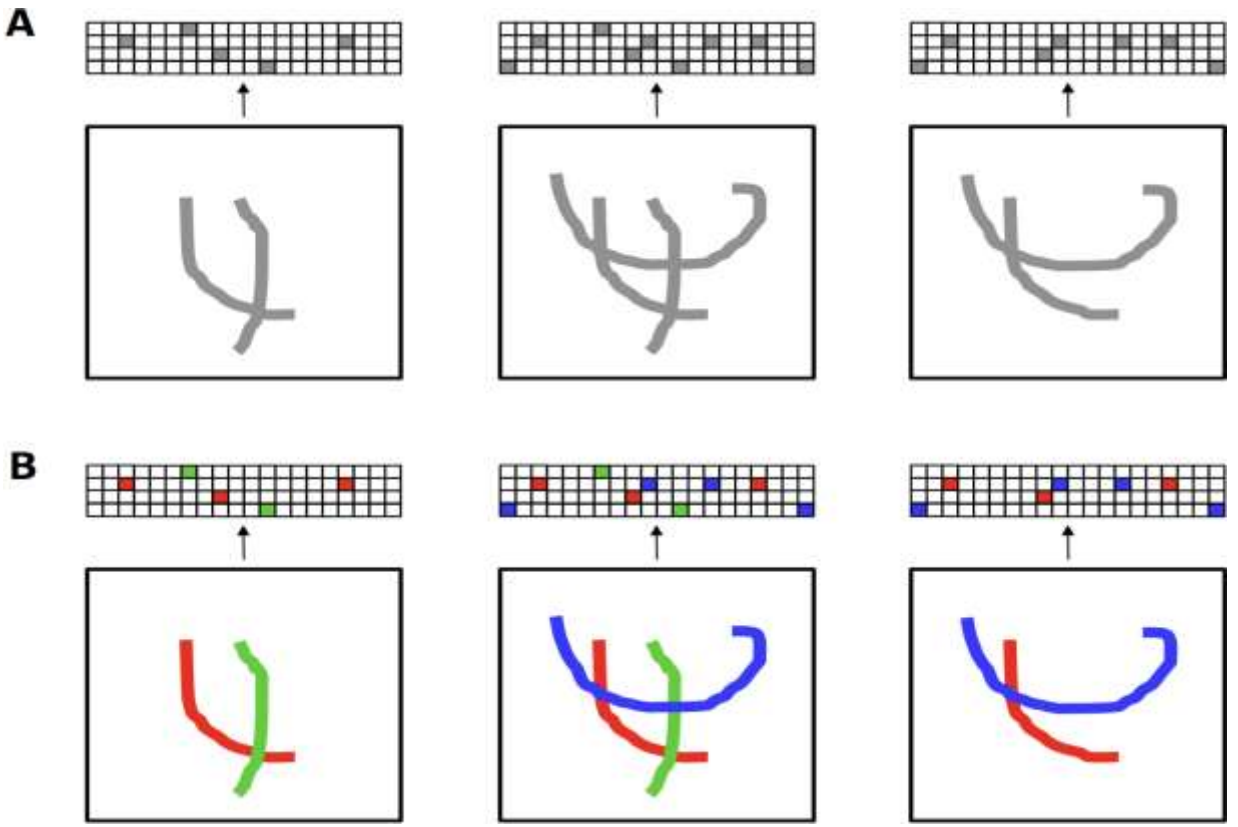


Ideal outcome

- As we add components or remove components, the output representation changes in unknown ways.
  - It’s virtually impossible to interpret how and why the representation is generated in the way it is.
- 1-to-1 correspondence (or close to 1-to-1) between high-level features and individual components in the code.
- When we are able to achieve this, we get very close to the system described next.
- Part A of the figure shows how the representation changes as we add and remove components
- Part B color-codes the correspondence between strokes and the components in the code.
- In this setup, it’s quite clear how and why the representation changes - the representation is very clearly the sum of the individual strokes in the image.

Code Layer Capacity

- While this is the ideal outcome, we'll have to think through what mechanisms we can leverage to enable this interpretability in the representation.
- The issue here is clearly the bottlenecked capacity of the code layer.
- But, increasing the capacity of the code layer alone is not sufficient.
- As, there is no mechanism to prevent each feature from affecting a large fraction of the components.
- If the features are more complex, the capacity of the code layer may be larger than the dimensionality of the input.
- The code layer has capacity that the model can perform a "copy" operation.



Sparsity in Autoencoders

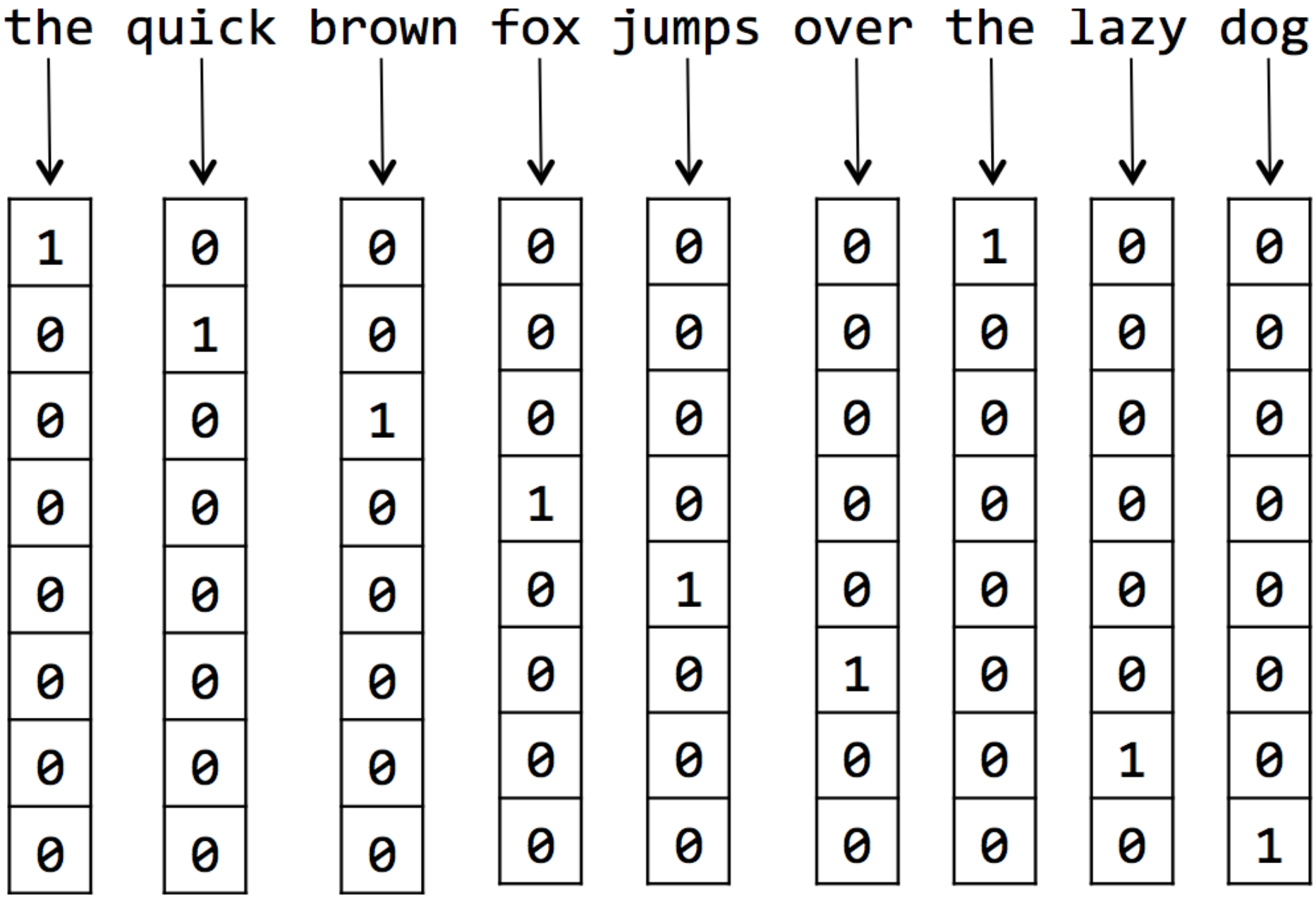
- We want the autoencoder to use very few components of the representation vector.
  - while still effectively reconstructing the input
  - similar to using regularization to prevent overfitting in Neural Net
- We'll achieve this by modifying the objective function with a sparsity penalty.
  - $E_{\text{Sparse}} = E + \beta \cdot \text{SparsityPenalty}$
  - β determines how strongly we favor sparsity.
- Measure of divergence compares the distribution of random variable (each component) and the distribution of a random variable whose mean is 0.
- A measure that is often used to this end is the Kullback-Leibler (often referred to as KL) divergence.
- *k-Sparse autoencoders* were shown to be just as effective as other mechanisms of sparsity despite being simple to implement and understand.

Concluding Autoencoders

- Further discussion on sparsity in autoencoders is covered by Ranzato et al.
- More recently, the theoretical properties and empirical effectiveness of introducing an intermediate function before the code layer that zeroes out all but *k* of the maximum activations in the representation were investigated by Makhzani and Frey.
- We've explored how we can use autoencoders to find strong representations of data points by summarizing their content.
- This mechanism of dimensionality reduction works well when the independent data points are rich and contain all of the relevant information pertaining to their structure in their original representation.
- In the next section, we'll explore strategies that we can use when the main source of information is in the context of the data point instead of the data point itself.

When Context Is More Informative than the Input Vector

- In dimensionality reduction, we generally have rich inputs which contain lots of noise on top of the core, structural information that we care about.
- We want to extract this underlying information.
- We have to ignore the variations and noise that are extraneous to this fundamental understanding of the data.
- Sometimes we have input representations that say little about the content.
- Here, our goal is not to extract information, but rather, to gather information from context to build useful representations.
  - Sounds too abstract to be useful at this point.
  - Example: Building models for language by finding a good way to represent individual words.



Generating one-hot vector representations for words using a simple document

one-hot vector

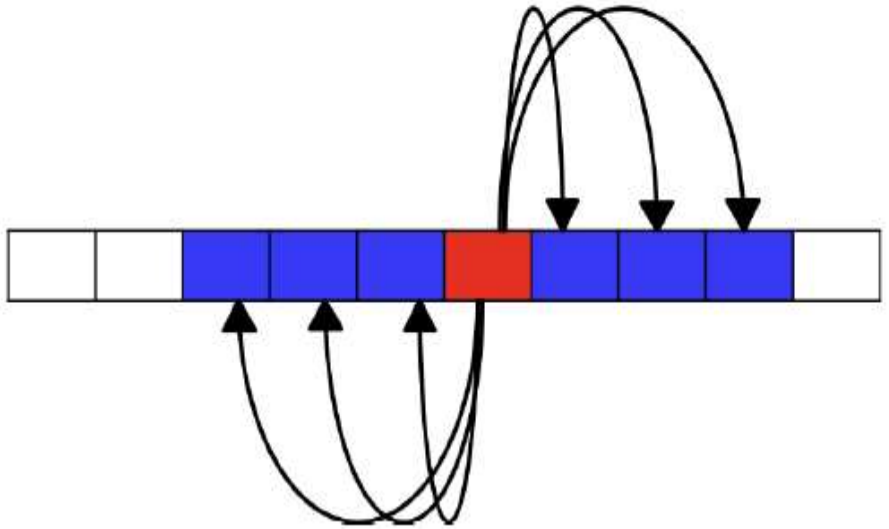
- The document has a vocabulary *V* with |V| words.
- We have |V| -dimensional representation vectors
- We associate each unique word with an index in this vector.
- To represent unique word *wi*, we set the *ith* component of the vector to be 1, and zero out all of the other components.
- This vectorization does not make similar words into similar vectors.
  - “jump” and “leap” have similar meanings?
  - when words are verbs or nouns or prepositions?
- Naive one-hot encoding of words to vectors doesn’t capture any of these info.
- We’ll need to discover these relationships and encode this information into a vector.
  - one way is by analyzing their surrounding context.

Example

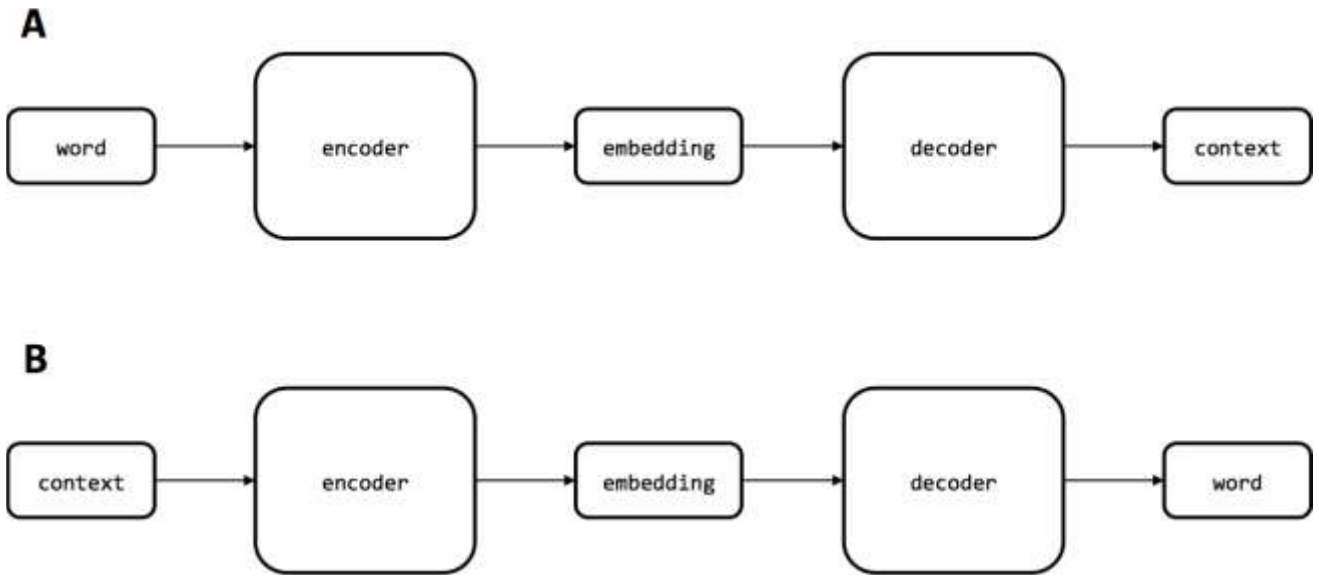
- Both words generally appear when a subject is performing the action over a direct object.
- We can draw conclusions about what the words “jumps” and “leaps” mean just by looking at the words around them.
- The words “jumps” and “leaps” should have similar vector representations because they are virtually interchangeable.

Brown fox jumps over the dog
The boy jumps over the fence
The man jumps over the pothole
The rabbit jumps over the tortoise
The company jumps over the hurdle

Brown fox leaps over the dog
The boy leaps over the fence
The man leaps over the pothole
The rabbit leaps over the tortoise
The company leaps over the hurdle



- We can use the same principles we used when building the autoencoder
  1. pass the target through an encoder network to create an embedding. Then decoder attempts to construct a word from the context.
  2. Do reverse: encoder takes a word from context as input, producing the target.



Word2Vec Framework

- A framework for generating word embeddings by Mikolov et al.
- Two strategies for generating embeddings similar to the two strategies for encoding context
  - *Continuous Bag of Words* (CBOW) model (much like strategy B for encoding context)
    - encoder creates embedding from the context and predict the target word.
    - Useful for smaller datasets
  - *Skip-Gram* model (inverse of CBOW)
    - It takes the target word as input and attempts to predict one of the words in the context.
    - Lets take a toy example to explore what the dataset for a Skip-Gram model looks like:
    - Consider the sentence: “the boy went to the bank.”
    - Break the sentence into (context, target) pairs => [([the, went], boy), ([boy, to], went), ([went, the], to), ([to, bank], the)]
    - Split each (context, target) pair into (input, output) pairs where the input is the target and the output is one of the words from the context => (boy, the) and (boy, went), (went, boy) and (went, to), (to, went) and (to,the), etc.
    - replace each word with its unique index $i \in \{0, 1, \ldots, |V|-1\}$ in the vocabulary.

Word2Vec Framework Encoder

- The structure of the encoder is surprisingly simple.
- It is a lookup table with *V* rows, where the *ith* row is the embedding corresponding to the *ith* vocabulary word.
- Encoder takes the index of the input word and output the appropriate row in the lookup table.
- This is an efficient operation.
- The operation can be represented as a product of the transpose of the lookup table and the one-hot vector representing the input word.

  tf.nn.embedding_lookup(params, ids, partition_strategy='mod', name=None, validate_indices=True)

  params is the embedding matrix, and ids is a tensor of indices we want to look up

Word2Vec Framework Decoder

- The decoder is slightly trickier.
- This is because we make some modifications for performance.
- The naive way to construct the decoder would be to attempt to reconstruct the one-hot encoding vector for the output, which we could implement with a run-of-the-mill feed-forward layer coupled with a softmax.
- The only concern is that it’s inefficient.
- That is because we have to produce a probability distribution over the whole vocabulary space.
- To reduce the number of parameters, Mikolov et al. used a strategy for implementing the decoder known as noise-contrastive estimation (NCE).

Noise-Contrastive Estimation (NCE)

- It uses the lookup table to find the embedding for the output, as well as for random vocabulary word not in the context of the input.
- It then employ binary logistic regression to take the input embedding and the embedding of the output or random selection.
- It outputs a value between 0 to 1 corresponding to the probability that the comparison embedding represents a vocabulary word present in the input's context.
- It then take the sum(probabilities corresponding to the non-context comparisons) – (probability corresponding to the context comparison).
- This value is the objective function that we want to minimize.
- In the optimal scenario where the model has perfect performance the value will be -1

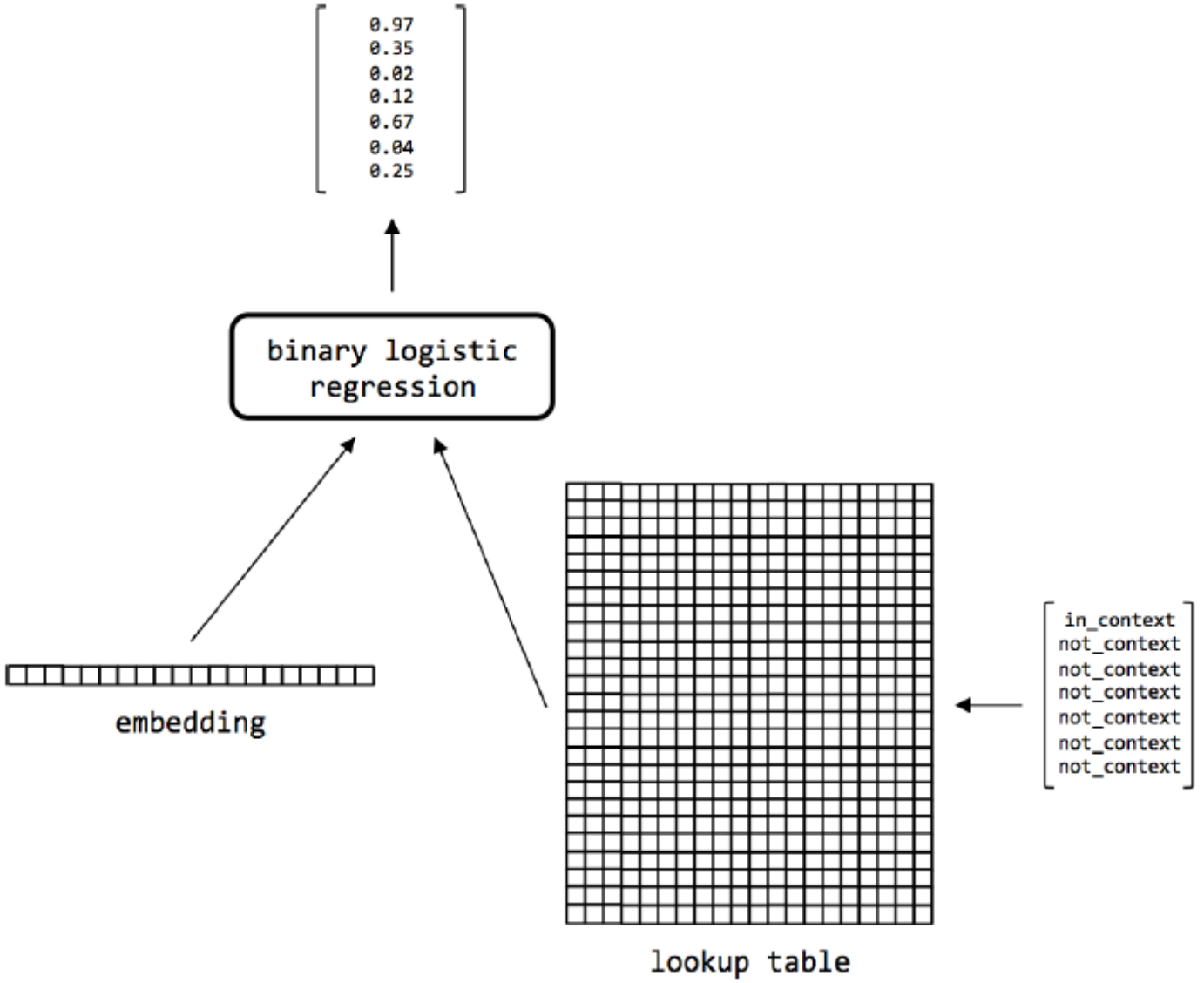
Implementing NCE in TensorFlow

tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=False, partition_strategy= 'mod', name='nce_loss')

- *weights* should have the same dimensions as the embedding matrix.
- *biases* should be a tensor with size equal to the vocabulary.
- The inputs are the results from the embedding lookup.
- num_sampled is the number of negative samples we use to compute the NCE
- num_classes is the vocabulary size

Word2Vec is not a deep machine learning model

- It thematically represents a strategy (finding embeddings using context) that generalizes to many deep learning models.
- When we'll learn sequence analysis, we'll see this strategy employed for generating vectors to embed sentences.
- Using Word2Vec embeddings instead of one-hot vectors to represent words will yield far superior results.
- Out of Syllabus: Implementing the Skip-Gram Architecture



Thank You