Deep Learning and Its Applications in Signal Processing

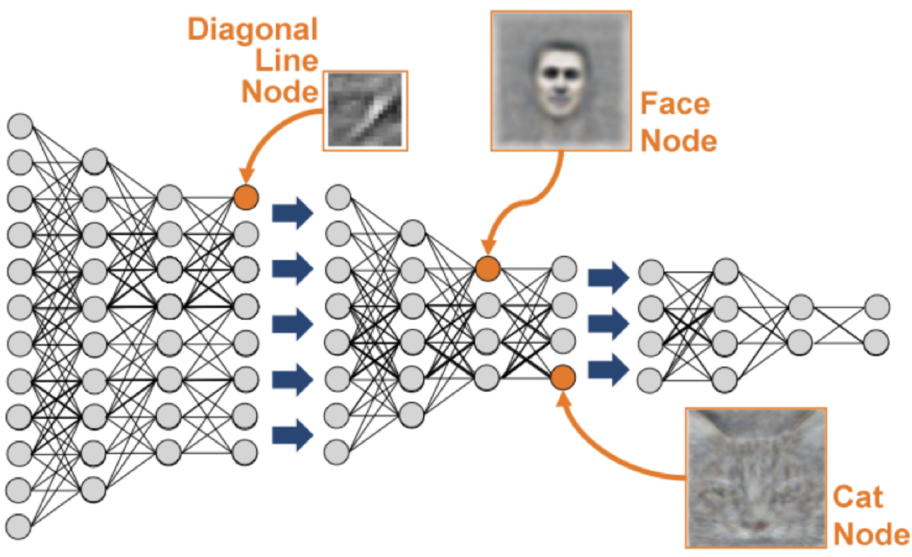
Lesson 1: Review of Deep Neural Networks

Liang Dong, ECE

Deep Neural Network

- Human brain contains about 86 billion nerve cells (neurons) – the "gray matter".
- It also contains billions of nerve fibers (axons and dendrites) – the "white matter".
- These neurons are connected by trillions of connections, or synapses.

Deep Neural Network



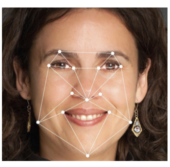
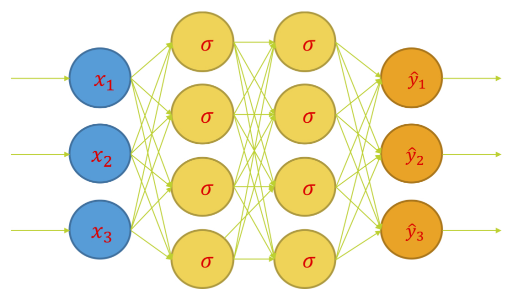


Figure: Perception of happiness

Deep Neural Network



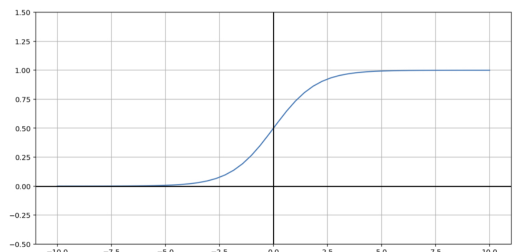
For example (the neural network shown left),

$$\hat{\mathbf{y}} = \text{softmax}\left(\sigma(\sigma(\mathbf{x}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3\right)$$

- The activation function $\sigma(\cdot)$ is nonlinear.
- A layered neural network is a highly nonlinear system that can model the complex reality.
- The activation function relates to logistic regression, e.g., using the "sigmoid" function $\sigma(x) = \frac{1}{1+e^{-x}}$.

Activation Function

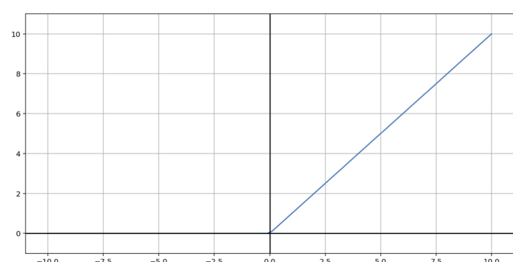
- The "sigmoid" function: $\sigma(x) = \frac{1}{1+e^{-x}}$



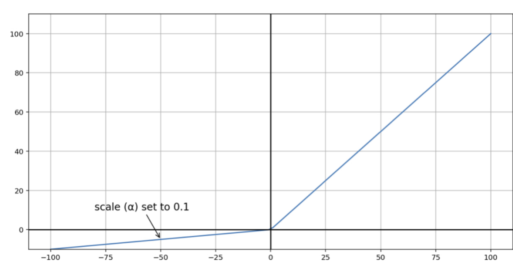
- Useful property of the sigmoid function: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

Activation Function

- Rectified Linear Unit (ReLU): $\mathrm{ReLU(x)} = \max(0, x)$

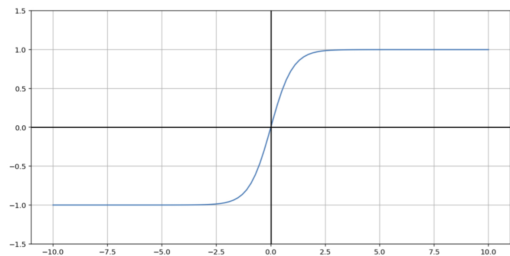


- "Leaky" Rectified Linear Unit: $\mathrm{LReLU}(x) = \max(\alpha x, x), 0 < \alpha < 1$



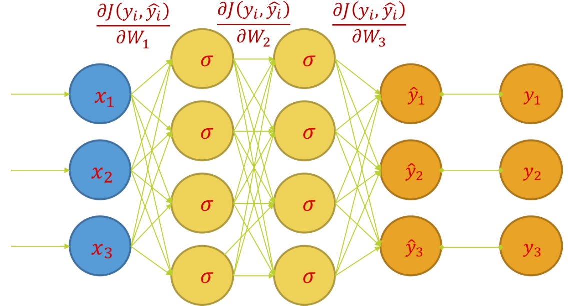
Activation Function

- Hyperbolic tangent function: $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x}-1}{e^{2x}+1}$



- $\tanh(0) = 0$, $\tanh(\infty) = 1$, $\tanh(-\infty) = -1$.

Training Through Backpropogation



Cost function (e.g., squared error):

$$J(y_i, \hat{y}_i) = |\hat{y}_i - y_i|^2$$

where y_i is the known correct output (target or label).

- Train the weights $\mathbf{W}$ by minimizing the cost function:

$$\min_{\mathbf{W}} J(\mathbf{y}, \hat{\mathbf{y}})$$

Training Through Backpropogation

- Gradient decent (to minimize the mean squared error)

$$w \leftarrow w - \alpha \nabla \frac{1}{2m} \sum_m \|\hat{\mathbf{y}} - \mathbf{y}\|^2$$

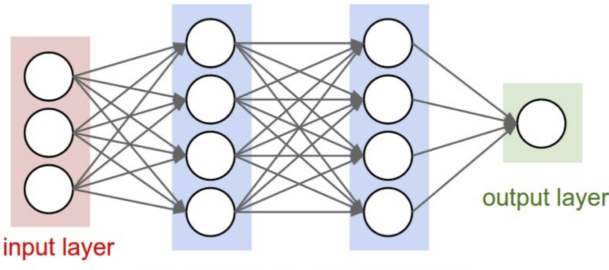
where m is the number of training samples and α is the learning rate.

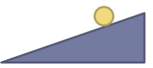
- Stochastic gradient decent

$$w \leftarrow w - \alpha \nabla \frac{1}{2} \|\hat{\mathbf{y}} - \mathbf{y}\|^2$$

Training Through Backpropogation

- Calculation of the early-layer gradients needs to use the weights of later layers.
- Backpropogation: Update the later-layer weights first then the early-layer weights.
- Vanishing gradient problem: The gradient becomes smaller and smaller at the early layers. (e.g., $\sigma'(x) = \sigma(x)(1 - \sigma(x)) \leq .25$.)





Parameter Estimator – Maximum Likelihood Estimation

- Maximum likelihood estimator is a preferred parameter estimator in terms of consistency and efficiency.

$$\begin{aligned}\boldsymbol{\theta}_{\mathrm{ML}} &= \arg\max_{\boldsymbol{\theta}} \Pi_{i=1}^{m} p_{\mathrm{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\mathbf{x}; \boldsymbol{\theta})\end{aligned}$$

- To minimize the dissimilarity between the empirical distribution $\hat{p}_{\mathrm{data}}$ and the model distribution p_{model}, we want to minimize the Kullback-Leibler (KL) divergence of the two distributions.

$$D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}} \| p_{\mathrm{model}}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}}[\log \hat{p}_{\mathrm{data}}(\mathbf{x}) - \log p_{\mathrm{model}}(\mathbf{x})]$$

Maximum Likelihood Estimation

- To minimize the KL divergence is to minimize the cross-entropy between the empirical distribution $\hat{p}_{\mathrm{data}}$ and the model distribution p_{model}.

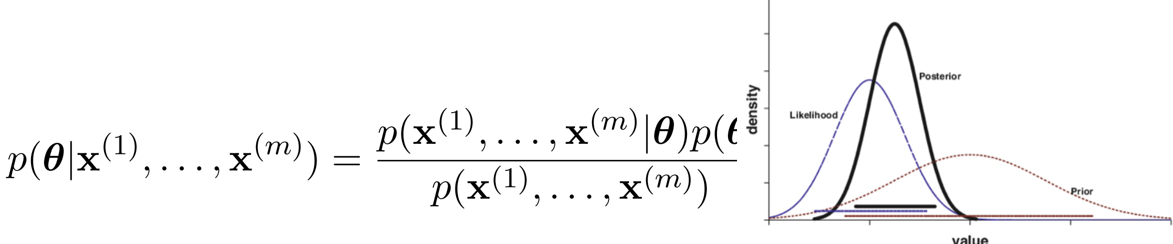
$$\text{minimize} - \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\mathbf{x})$$

- Maximum likelihood estimation is equivalent to minimization of the cross-entropy between the distributions.

Parameter Estimation – Bayesian Estimation

- Bayesian estimator makes prediction using a full distribution over parameter $\boldsymbol{\theta}$.
- Bayesian estimator has an influence from the prior distribution $p(\boldsymbol{\theta})$, which expresses a preference for the model.
- With a set of data, we can recover the effect of data on our belief about $\boldsymbol{\theta}$.

$$p(\boldsymbol{\theta}|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}) = \frac{p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})}$$



Bayesian Estimation

- With a set of data, the predicted distribution over the next data sample is

$$p(\mathbf{x}^{(m+1)}|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}) = \int p(\mathbf{x}^{(m+1)}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})d\boldsymbol{\theta}$$

- Bayesian estimator generalizes better when limited training data is available.
- Bayesian estimator suffers from high computational cost when training data set is large.

Maximum *a posteriori* (MAP) Estimation

- In stead of the full Bayesian posterior distribution over the parameter $\boldsymbol{\theta}$, we want a single point estimate.
- Maximum *a posteriori* (MAP) estimator chooses the point of maximal posterior probability:

$$\begin{aligned}\boldsymbol{\theta}_{\text{MAP}} &= \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}) \\ &= \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\end{aligned}$$

- Many regularized estimation strategies can be interpreted as making the MAP approximation to Bayesian inference. The regularization consists of adding an extra term to the objective function that corresponds to $\log p(\boldsymbol{\theta})$.

Optimizer – Stochastic Gradient Descent

Models trained with gradient descent – To find a very low value of the cost function (may not be global or even local minimum)

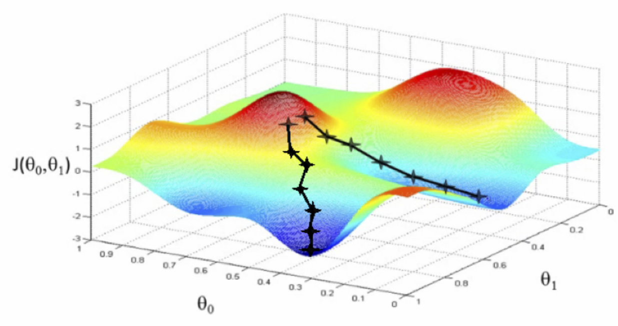


Figure: The gradient decent process may end up in two local minimums.

$$\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} \text{Loss}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}; \boldsymbol{\theta})$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \mathbf{g}$$

Stochastic Gradient Descent

- Motivation: Large training sets are more computationally expensive.
- The gradient is an expectation. The expectation can be approximately estimated with a small set of samples.
- Stochastic gradient descent (SGD) is an extension of the gradient descent algorithm. Calculate the gradient for one new sample and take a step in that direction.

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} \text{Loss}(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta})$$

- Compromise approach – Minibatch with size m' ($m' < m$):

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} \text{Loss}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}; \boldsymbol{\theta})$$

Stochastic Gradient Descent

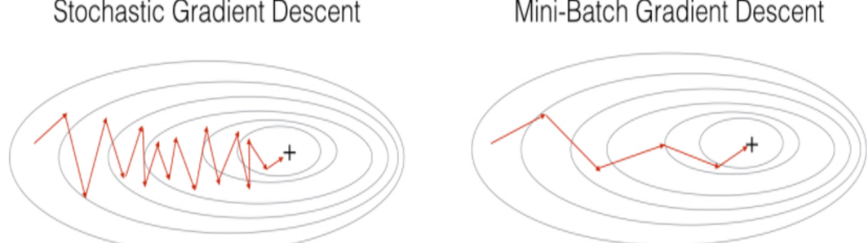


Figure: Comparison of stochastic gradient decent and minibatch gradient decent.

Model Evaluation Metrics – Classification Model

- Classification Accuracy

$$\text{Accuracy} = \frac{\text{Number of corrected predictions}}{\text{Total number of predictions}}$$

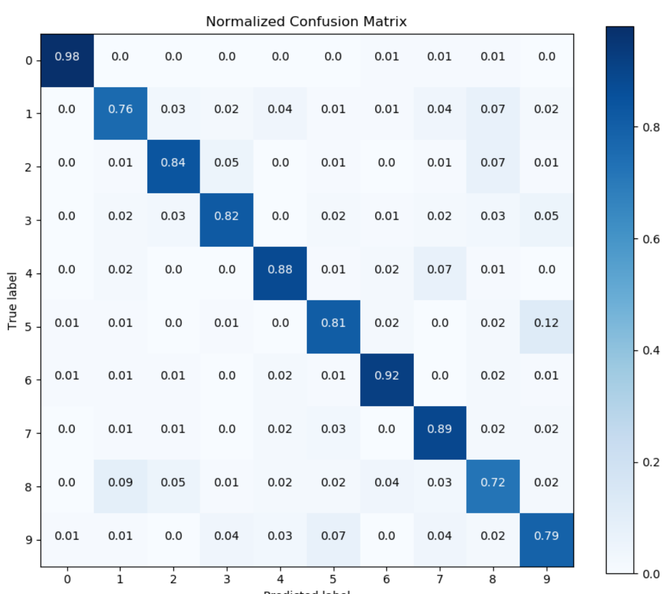
- Logarithmic Loss. It works well for multi-class classification by penalising the false classifications.

$$\text{Log Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij}$$

where y_{ij} indicates whether sample i belongs to class j and p_{ij} is the probability of sample i belonging to class j.

Model Evaluation Metrics

- Confusion Matrix. A confusion matrix is an $N \times N$ matrix, where N is the number of classes being predicted.



Model Evaluation Metrics

- Confusion Matrix for a binary classification problem. $N = 2$.

	Model Positive	Model Negative
Target Positive	True Positive (TP)	False Negative (FN)
Target Negative	False Positive (FP)	True Negative (TN)

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}}$$

Model Evaluation Metrics

- Confusion Matrix for a binary classification problem. $N = 2$.

	Model Positive	Model Negative
Target Positive	True Positive (TP)	False Negative (FN)
Target Negative	False Positive (FP)	True Negative (TN)

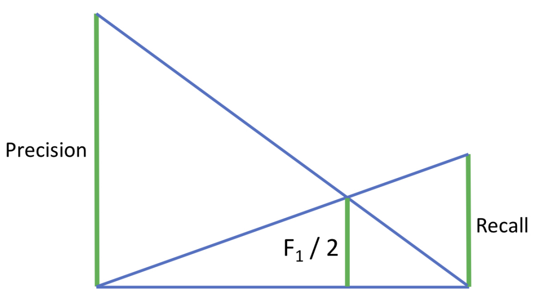
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Model Evaluation Metrics

- Confusion Matrix for a binary classification problem. $N = 2$.

	Model Positive	Model Negative
Target Positive	True Positive (TP)	False Negative (FN)
Target Negative	False Positive (FP)	True Negative (TN)

$$\text{Recall} = \text{True Positive Rate} = \text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$



$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Model Evaluation Metrics

- Evaluating models on an imbalanced data set.

First Model

Target\Model	A	B	C
A	150	30	20
B	0	9	1
C	2	0	8

$$\text{Accuracy} = 0.76$$
$$F_1 \text{ Score} = 0.62$$

"Better model" according to F_1 score

Second Model

Target\Model	A	B	C
A	198	2	0
B	9	1	0
C	4	4	2

$$\text{Accuracy} = 0.91$$
$$F_1 \text{ Score} = 0.53$$

Model Evaluation Metrics

- Confusion Matrix for a binary classification problem. $N = 2$.

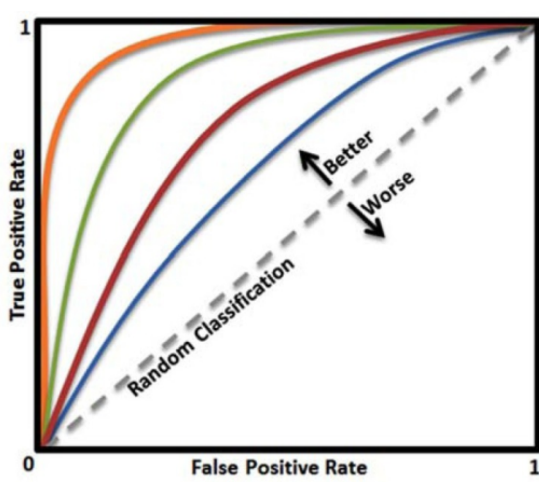
	Model Positive	Model Negative
Target Positive	True Positive (TP)	False Negative (FN)
Target Negative	False Positive (FP)	True Negative (TN)

$$\text{Specificity} = \frac{\text{TN}}{\text{FP} + \text{TN}}$$

$$\text{False Positive Rate} = 1 - \text{Specificity} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Model Evaluation Metrics

- AUC–ROC: Area Under the receiver operating characteristic (ROC) curve



True Positive Rate = Sensitivity. False Positive Rate = 1 - Specificity.

Model Evaluation Metrics – Regression Model

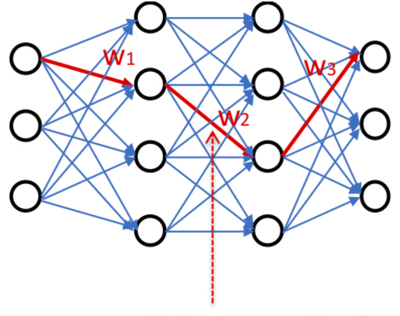
- Mean Absolute Error

$$\text{Error} = \frac{1}{N}\sum_{j=1}^{N} |y_j - \hat{y}_j|$$

- Root Mean Squared Error

$$\text{Error} = \sqrt{\frac{1}{N}\sum_{j=1}^{N} (y_j - \hat{y}_j)^2}$$

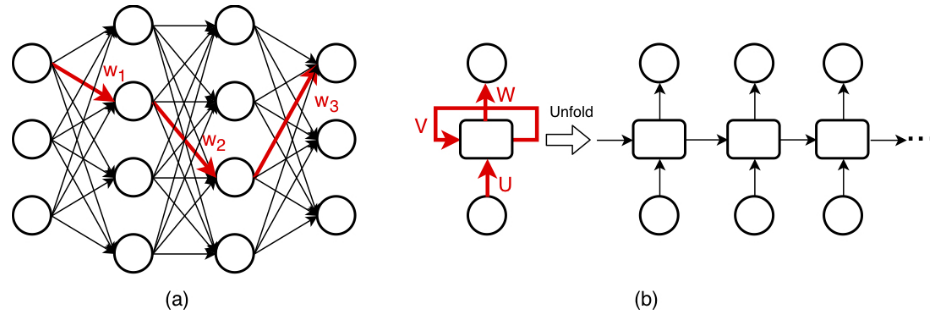
Credit Assignment Path and Depth of Learning



Credit Assignment Path of feedforward neural network e.g., Depth = 3

- The credit assignment path is a chain of causal links, some of which have modifiable weights.
- Of a credit assignment path, the number of causal links with modifiable weights is the depth.
- In a neural network, the maximum depth of all credit assignment paths is the depth of learning.

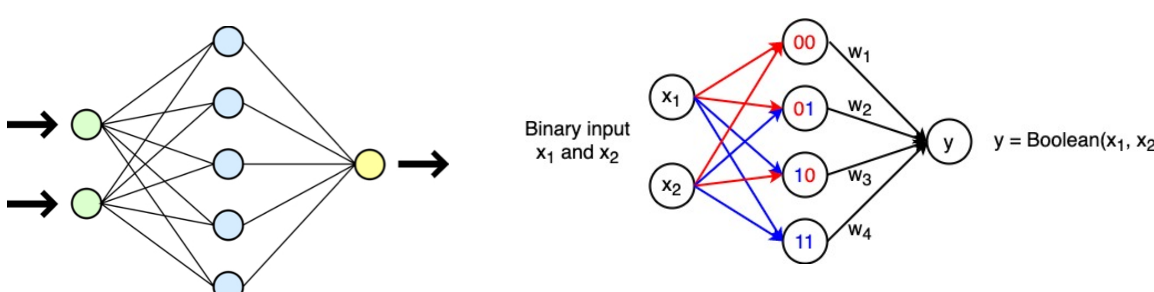
Credit Assignment Path and Depth of Learning



- (a) Credit assignment path of a feedforward neural network. In this example, depth is three.
- (b) Credit assignment path of a recurrent neural network whose depth can be unlimited.

Deep Learning vs. Shallow Learning

- A learning system with a depth of two has proven to be a universal approximator.
- A neural network with one hidden layer can represent any bounded continuous function (to arbitrary ϵ) or any Boolean function (exactly).



$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon$

With d input binary values, it may need 2^d nodes in the hidden layer.

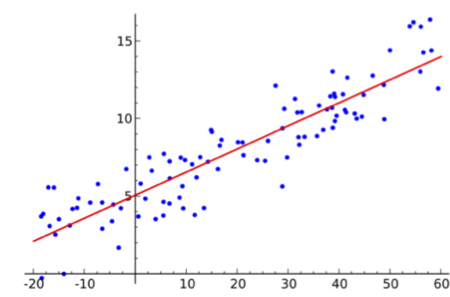
Deep Learning vs. Shallow Learning

- Modern deep learning systems typically have learning depths that are numbered in tens and hundreds. It is difficult to determine the exact depth that distinguishes deep learning from shallow learning.
- A greater depth – Not only does it better extract features from the input data, but it also reduces system coefficients for the same learning performance.
- Sometimes, good learning performance involves selecting a network structure that matches the particular data structure.

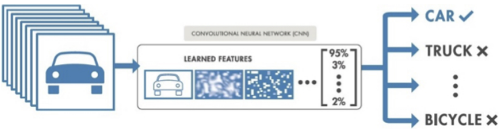
Supervised Learning

Supervised Learning: Observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, it predicts $\mathbf{y}$ from $\mathbf{x}$ by estimating $p(\mathbf{y}|\mathbf{x})$.

Regression: Learning algorithm output is $f : \mathbb{R}^n \to \mathbb{R}$.

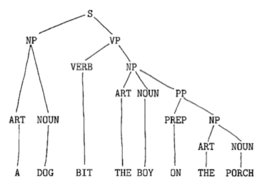


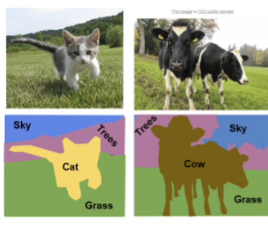
Classification: Learning algorithm output is $f : \mathbb{R}^n \to \{1, 2, \ldots, k\}$, or a probability distribution over the classes.

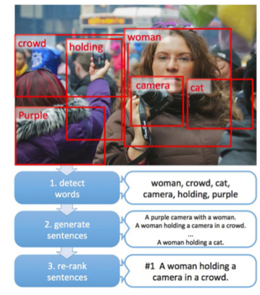


Supervised Learning

Structured Output: Sentence parsing, image segmentation, object detection, image captioning, transcription, language translation, etc.



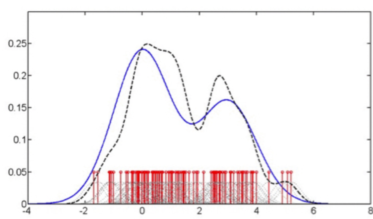


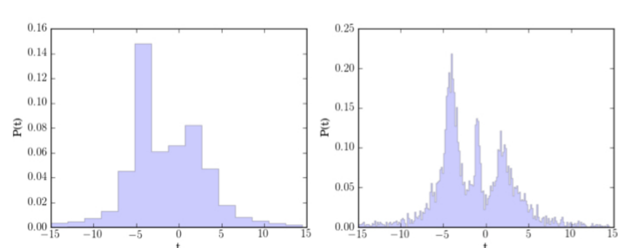


Unsupervised Learning

Unsupervised Learning: Observing several examples of a random vector $\mathbf{x}$, it explicitly or implicitly learns $p(\mathbf{x})$ or some properties of this distribution.

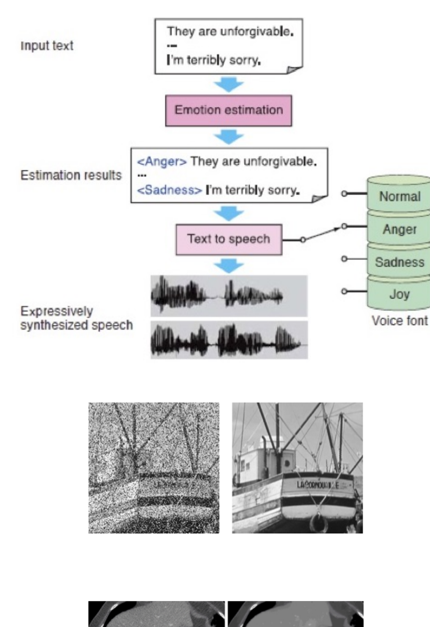
Density Estimation (or Prob. Mass Function Estimation): Learning the probability distribution that generated a dataset. Learning useful properties of the structure of the dataset. Dividing the dataset into clusters of similar examples (clustering).





Unsupervised Learning

Synthesis: Generating new samples that are similar to the training data. For example, speech synthesis.



Denoising: Generating a clean example from a corrupted example with unknown corruption process.

The Curse of Dimensionality

Curse of Dimensionality – The number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases.

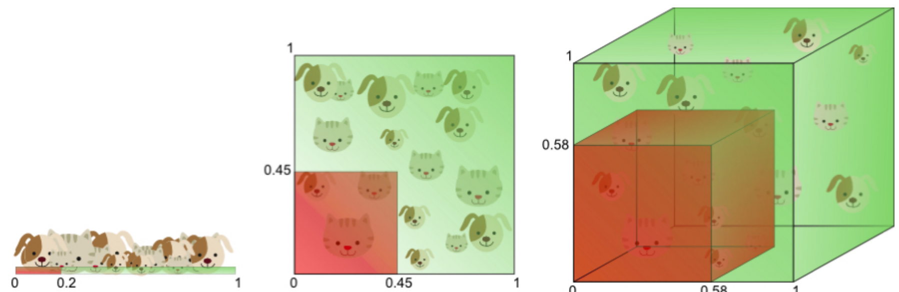


Figure: The amount of training data needed to cover 20% of the feature range grows exponentially with the number of dimensions.

Vincent Spruyt, "The Curse of Dimensionality in classification".

The Curse of Dimensionality: Deep-Layered Representation

- A very large number of regions, such as $O(2^k)$, can be defined with $O(k)$ examples, so long as we introduce some dependencies between the regions through additional assumptions about the underlying data-generating distribution.
- The core idea of deep learning:
  - It is assumed that the data is generated by a combination of factors or features, possibly at multiple levels of the hierarchy.
  - The advantages conferred by the use of deep distributed representations counter the challenges posed by the curse of dimensionality.

Manifold Learning

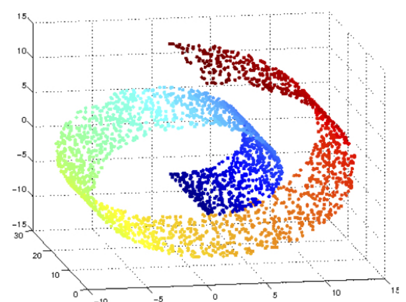
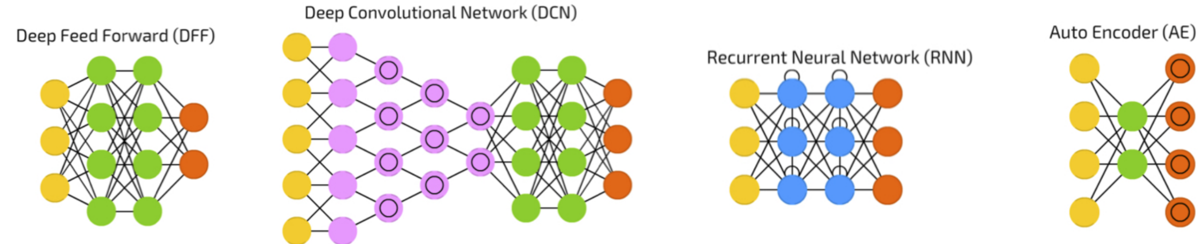


Figure: A manifold is a connected region in high-dimensional space.

- A connected set of points that can be approximated well by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space.
- Interesting inputs occur only along a collection of manifolds containing a small subset of points.
- Interesting variations in the output of the learned function occur only along directions that lie on the manifolds (or from one manifold to another).
- Manifold Learning, because the probability distribution of images, sounds or text strings in real life is highly concentrated.

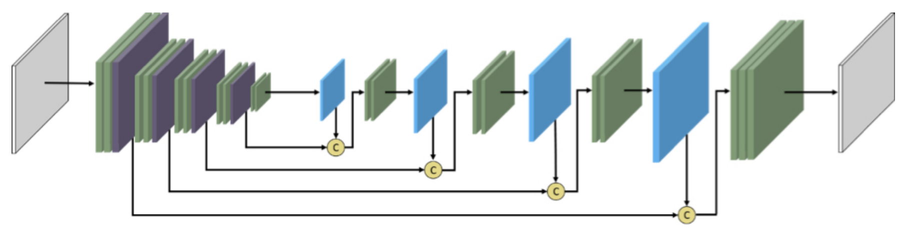
Architecture Design for Deep Neural Network

- Most neural networks arrange layers of units in a chain structure, with each layer being a function of the layer that precedes it.
- In these chain-based architectures, the main considerations are choosing the depth of the network and the width of each layer.
- Another key consideration of architecture design is how to connect a pair of layers to each other. e.g., full connection vs. sparse connection.
- Deeper networks typically use fewer units per layer, use fewer parameters, and are often able to generalize to test sets.



Architecture Design for Deep Neural Network

- In general, the layers need not be connected in a chain, even though this is the most common practice.
- Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers near the input.



Architecture Design for Deep Neural Network

- By adding more layers and more units within a layer, one can use a deep neural network to represent functions of increasing complexity.
- The ideal network architecture for a task is found with experiments guided by monitoring the validation set error.
- Specialized architectures have been developed for specific tasks. e.g.,

Deep Convolutional Network (DCN)

Figure: CNN for computer vision.

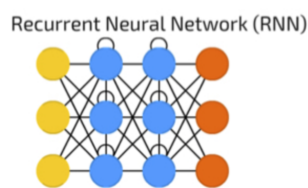


Figure: RNN for sequence processing.

Steps to Establish A Deep Learning Algorithm

Establish a deep learning algorithm (for supervised learning):

1. Specify a dataset – training data, verification data, and test data.
2. Design model architecture of a deep neural network.
3. Model parameter training:
    - Define a loss function according to parameter estimator, e.g., the negative log-likelihood. Loss function may include additional terms such as regularization terms.
    - Use an iterative numerical optimizer for gradient-based learning.
4. Evaluation – Cross validation and testing of the trained model.

Deep Learning Algorithm Examples with Keras

- Keras is a high-level neural network API, written in Python and capable of running on top of TensorFlow TensorFlow, CNTK, or Theano theano.
- Keras is a deep learning library that allows for easy and fast prototyping through user friendliness, modularity, and extensibility
- It can run seamlessly on CPU and GPU.

Example: Feedforward Network – Multilayer Perceptron

```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```

Example: Feedforward Network – Multilayer Perceptron

```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```

- Fully connected layer: 10 input-layer neurons and 20 hidden-layer1 neurons.

Example: Feedforward Network – Multilayer Perceptron

```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```

- Fully connected layer: 20 hidden-layer1 neurons and 20 hidden-layer2 neurons.

Example: Feedforward Network – Multilayer Perceptron

```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```

- Fully connected layer: 20 hidden-layer2 neurons and 10 output-layer neurons.

Example: Feedforward Network – Multilayer Perceptron


```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```


- Activation function – a fixed nonlinear function that is after an affine transformation.
- Rectified Linear Unit (ReLU) by default.

Example: Feedforward Network – Multilayer Perceptron

```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Dense(20, activation=tf.nn.relu), input_shape=(10,))
model.add(Dense(20, activation=tf.nn.relu))
model.add(Dense(10, activation=tf.nn.softmax))
```

- Activation function – Softmax.

Example: Feedforward Network – Multilayer Perceptron

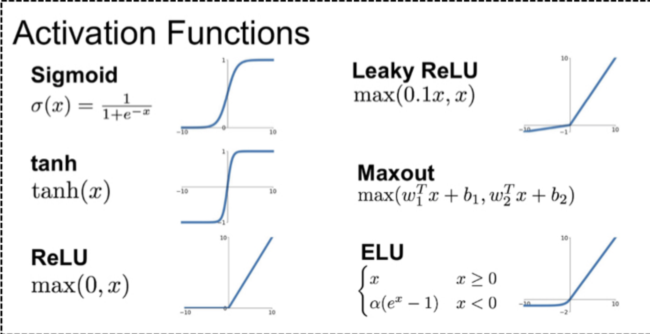
```
import tensorflow as tf

# Importing the required Keras modules containing model and layers
from tensorflow.keras.layers import Dense

# Creating a Functional API
inputs = keras.Input(shape=(10,))
x = Dense(20, activation='relu')(x)
x = Dense(20, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
```

Activation Functions of Hidden Units



- Rectified linear units (ReLU) are default for hidden units.
- Generalization: Absolute value rectification, leaky ReLU, parametric ReLU
- Maxout units
- Logistic Sigmoid
- Hyperbolic Tangent
- Linear hidden units
- Softmax units
- Radial basis function
- Softplus

Output Units

- Linear Units for Gaussian output distributions
- Sigmoid Units for Bernoulli output distributions (to ensure that there is always a strong gradient when the model has the wrong answer)
- Softmax Units for Multinoulli output distributions
- Gaussian mixture outputs of mixture density networks

- The choice of loss (cost) function is tightly coupled with the choice of output unit.

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```


Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```


- 2D convolutional layer: kernel size 3×3, 28 filters (depth 28).

Example: Convolutional Neural Network

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid',
                    data_format=None, dilation_rate=(1, 1), activation=None,
                    use_bias=True, kernel_initializer='glorot_uniform',
                    bias_initializer='zeros', kernel_regularizer=None,
                    bias_regularizer=None, activity_regularizer=None,
                    kernel_constraint=None, bias_constraint=None)
```

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

- 2D max pooling layer: pool size 2×2.

Example: Convolutional Neural Network

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None,
                          padding='valid', data_format=None)
```

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

- Flattening the 2D arrays for fully connected layers.

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

- Fully connected layers.
- Out-layer 128 neurons. Activation function – ReLU.

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```


- Dropout rate $= 0.2$. That is the fraction of the units to drop.

Example: Convolutional Neural Network

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

- Fully connected layers.
- Out-layer 10 neurons. Activation function – Softmax.

Training the Model

```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

Training the Model

```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

Gradient-Based Learning

- Most loss functions of neural networks are nonconvex.
- (Stochastic) gradient descent has no convergence guarantee.
- Drive the cost function to a very low value – may not have convergence guarantees
- Sensitive to the initial parameters
- Initialize all weights to small random values, biases to zero or small positive values

Training the Model


```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```


Optimizers in Keras:

- SGD: Stochastic gradient descent optimizer
- RMSProp: RMSProp optimizer (good for RNN)
- Adagrad and Adadelta optimzers
- Adam: Adam optimizer [1]
- Adamax: A variant of Adam
- Nadam: Nesterov Adam optimizer

[1] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic

Training the Model

```
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

Loss (Cost) Function

- Principle of maximum likelihood – Use the cross-entropy between the training data and the model's predictions as the cost function.
- The total cost function used to train a neural network will often combine one of the primary cost functions with a regularization term.

$$L(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x})$$

Training the Model


```
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```


Loss (Cost) Function

- If is Gaussian, we have the mean squared error cost.

$$L(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x};\boldsymbol{\theta})\|^2 + \text{const}$$

Training the Model

```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

Loss (Cost) Function

- Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients.
- The cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(\mathbf{y} \mid \mathbf{x})$.

Training the Model


```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```


Loss (Cost) Function in Keras

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh

Training the Model

```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

Loss (Cost) Function in Keras

- categorical_crossentrophy
- sparse_categorical _crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Training the Model

```
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train,y_train,epochs=10)
```

A Metric is a function to judge the performance of the model.

- binary_accuracy
- categorical_accuracy
- sparse_categorical_accuracy
- top_k_categorical_accuracy
- sparse_top_k_categorical _accuracy

- Custom Metrics

Keras Model Attributes and Methods

- model.layers is a flattened list of the layers comprising the model.
- model.inputs is the list of input tensors of the model.
- model.outputs is the list of output tensors of the model.
- model.summary() prints a summary representation of your model.
- model.get_weights() returns a list of all weight tensors in the model, as Numpy arrays.
- model.set_weights(weights) sets the values of the weights of the model, from a list of Numpy arrays.