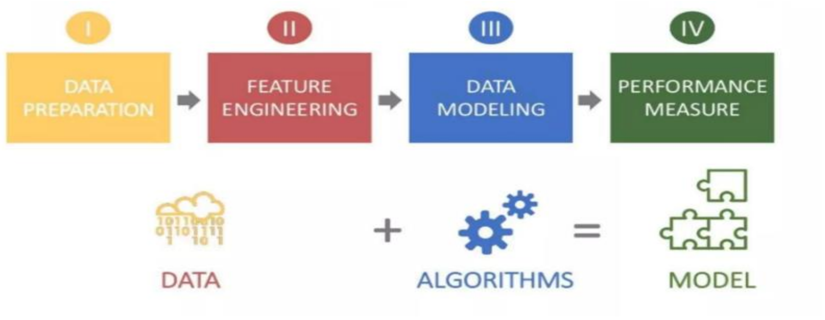
UNIT – 5

Introduction:

- Successfully applying deep learning techniques requires more than just a good knowledge of what algorithms exist and the principles that explain how they work.
- A good machine learning practitioner also needs to know how to choose an algorithm for a particular application and how to monitor and respond to feedback obtained from experiments in order to improve a machine learning system.
- During day to day development of machine learning systems, practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model.
- This may give the impression that the most important ingredient to being a machine learning expert is knowing a wide variety of machine learning techniques and being good at different kinds of math.
- We recommend the following practical design process:
    - Determine your goals—what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the problem that the application is intended to solve.
    - Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
    - Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether it is due to overfitting, underfitting, or a defect in the data or software.
    - Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.
- The purpose of this application is to add buildings to Google Maps. Street View cars photograph the buildings and record the GPS coordinates associated with each photograph.

Performance Metrics:

- After doing the usual Feature Engineering, Selection, and of course, implementing a model and getting some output in forms of a probability or a class, the next step is to find out how effective is the model based on some metric using test datasets.
- The metrics that you choose to evaluate your machine learning model is very important. Choice of metrics influences how the performance of machine learning algorithms is measured and compared.
- Performance metrics are a part of every machine learning pipeline. They tell you if you're making progress, and put a number on it. All machine learning models, whether it's linear regression, or a SOTA technique like BERT, need a metric to judge performance.
- Every machine learning task can be broken down to either *Regression* or *Classification*, just like the performance metrics. There are dozens of metrics for both problems, but we're gonna discuss popular ones along with what information they provide about model performance. It's important to know how your model sees your data!
- If you ever participated in a Kaggle competition, you probably noticed the evaluation section. More often than not, there's a metric on which they judge your performance.
- **Metrics are different from loss functions**. Loss functions show a measure of model performance. They're used to train a machine learning model (using some kind of optimization like Gradient Descent), and they're usually differentiable in the model's parameters.
- Metrics are used to monitor and measure the performance of a model (during training and testing), and don't need to be differentiable.
- However, if, for some tasks, the performance metric is differentiable, it can also be used as a loss function (perhaps with some regularizations added to it), such as MSE.



- Different metrics used:
  - Confusion Matrix
  - Accuracy
  - Precision
  - Recall or Sensitivity
  - Specificity
  - F1 Score
  - Log Loss
  - Area under the curve (AUC)
  - MAE - Mean Absolute Error
  - MSE - Mean Squared Error
- **Confusion Matrix:**
  - Just opposite to what the name suggests, confusion matrix is one of the most intuitive and easiest metrics used for finding the correctness and accuracy of the model. It is used for Classification problem where the output can be of two or more types of classes.
  - Let's say we are solving a classification problem where we are predicting whether a person is having cancer or not.
  - Let's give a label of to our target variable:
    1. When a person is having cancer 0: When a person is NOT having cancer.

       Alright! Now that we have identified the problem, the confusion matrix, is a table with two dimensions ("Actual" and "Predicted"), and sets of "classes" in both dimensions. Our Actual classifications are columns and Predicted ones are Rows.

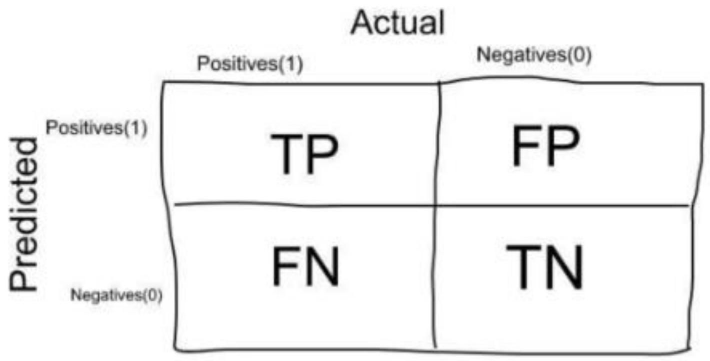
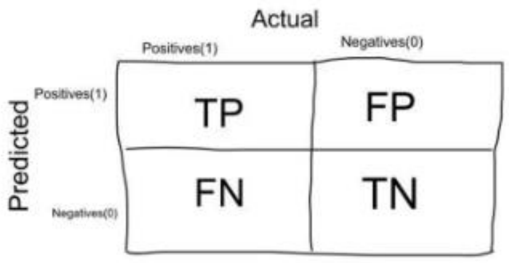
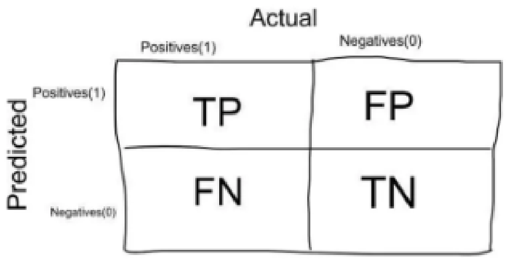


Fig. 1: Confusion Matrix

- The Confusion matrix in itself is not a performance measure as such, a lot of the performance metrics are based on Confusion Matrix and the numbers inside it.



- True Positives (TP) - True positives are the cases when the actual class of the data point was 1(True) and the predicted is also 1(True). Ex: The case where a person is actually having cancer(1) and the model classifying his case as cancer(1) comes under True positive
- True Negatives (TN) - True negatives are the cases when the actual class of the data point was 0(False) and the predicted is also 0(False)
  Ex: The case where a person NOT having cancer and the model classifying his case as Not cancer comes under True Negatives.

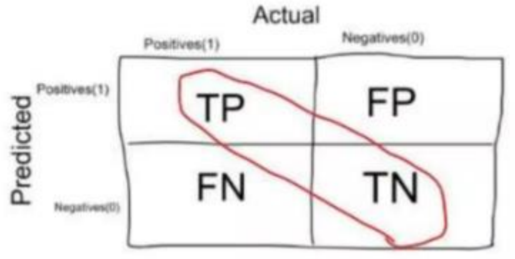


- False Positives (FP) - False positives are the cases when the actual class of the data point was 0(False) and the predicted is 1(True).

False is because the model has predicted incorrectly and positive because the class predicted was a positive one. (1)
Ex: A person NOT having cancer and the model classifying his case as cancer comes under False Positives.

- False Negatives (FN) - False negatives are the cases when the actual class of the data point was 1(True) and the predicted is O(False). False is because the model has predicted incorrectly and negative because the class predicted was a negative one. (0)
  Ex: A person having cancer and the model classifying his case as No-cancer comes under False Negatives.
- The ideal scenario that we all want is that the model should give 0 False Positives and 0 False Negatives.But that's not the case in real life as any model will NOT be 100% accurate most of the times.
- When to minimize what?
  - We know that there will be some error associated with every model that we use for predicting the true class of the target variable. This will result in False Positives and False Negatives
  - There's no hard rule that says what should be minimised in all the situations. It purely depends on the business needs and the context of the problem you are trying to solve. Based on that, we might want to minimise either False Positives or False negatives.

- **Accuracy:**
  - Accuracy in classification problems is the number of correct predictions made by the model over all kinds predictions made.
  - In the Numerator, are our correct predictions (True positives and True Negatives) (marked as red in the fig above) and in the denominator, are the kind of all predictions made by the algorithm (right as well as wrong ones).

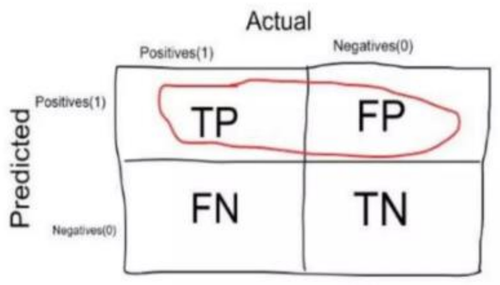


$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}$$

- **When to use Accuracy:**
    - Accuracy is a good measure when the target variable classes in the data are nearly balanced.
      Eg: 60% classes in our fruits images data are apple and 40% are oranges.
    - A model which predicts whether a new image is Apple or an Orange, 97% of times correctly is a very good measure in this example.

- **When not to use Accuracy:**
    - Accuracy should never be used as a measure when the target variable classes in the data are a majority of one class.
      Eg: In our cancer detection example with 100 people, only 5 people has cancer.
    - Let's say our model is very bad and predicts every case as No Cancer. In doing so, it has classified those 95 non-cancer patients correctly and 5 cancerous patients as Non-cancerous. Now even though the model is terrible at predicting cancer, The accuracy of such a bad model is also 95%.

- <u>**Precision:**</u>
    - Precision is a measure that tells us what proportion of patients that we diagnosed as having cancer, actually had cancer. The predicted positives (People predicted as cancerous are TP and FP) and the people actually having a cancer are TP.

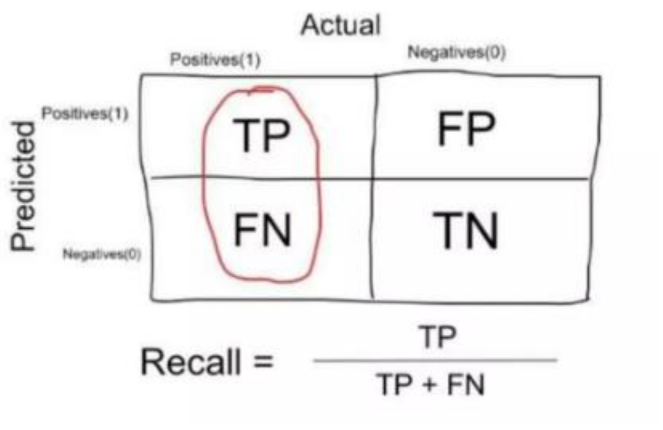


$$\text{Precision} = \frac{TP}{TP + FP}$$

    - In our cancer example with 100 people, only 5 people have cancer. Let's say our model is very bad and predicts every case as Cancer.

  - Since we are predicting everyone as having cancer, our denominator (True positives and False Positives) is 100 and the numerator, person having cancer and the model predicting his case as cancer is 5. So in this example, we can say that Precision of such model is 5%.

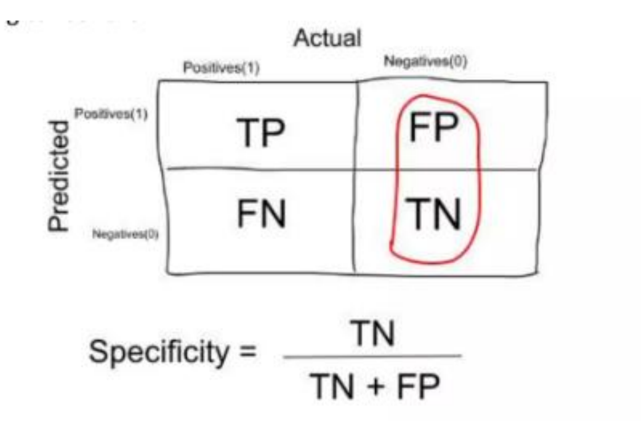
- **<u>Recall or Sensitivity:</u>**
  - Recall is a measure that tells us what proportion of patients that actually had cancer was diagnosed by the algorithm as having cancer.
  - The actual positives (People having cancer are TP and FN) and the people diagnosed by the model having a cancer are TP. (Note: FN is included because the Person actually had a cancer even though the model predicted otherwise).



$$\text{Recall} = \frac{TP}{TP + FN}$$

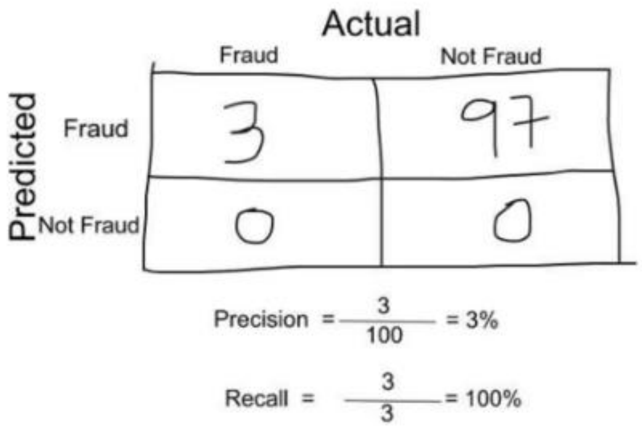
  - Ex: In our cancer example with 100 people, 5 people actually have cancer. Let's say that the model predicts every case as cancer. So our denominator(True positives and False Negatives) is 5 and the numerator, person having cancer and the model predicting his case as cancer is also 5(Since we predicted 5 cancer cases correctly). So in this example, we can say that the Recall of such model is 100%. And Precision of such a model(As we saw above) is 5%

- **<u>Specificity:</u>**
  - Specificity is a measure that tells us what proportion of patients that did NOT have cancer, were predicted by the model as non-cancerous. The actual negatives and the people diagnosed by us not having cancer are TN.



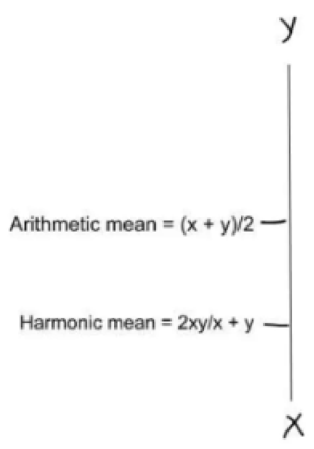
- **<u>Specificity is the exact opposite of Recall.</u>**
  Ex: In our cancer example with 100 people, 5 people actually have cancer. Let's say that the model predicts every
  case as cancer.
  So our denominator(False positives and True Negatives) is 95 and the numerator, person not having cancer and the
  model predicting his case as no cancer is 0 (Since we predicted every case as cancer). So in this example, we can that
  that Specificity of such model is 0%.

- **<u>F1 Score:</u>**
  - We don't really want to carry both Precision and Recall in our pockets every time we make a model for solving a classification problem. So it's best if we can get a single score that kind of represents both Precision(P) and Recall(R).
  - One way to do that is simply taking their arithmetic mean. i.e (P + R) / 2 where P is Precision and R is Recall. But that's pretty bad in some situations.
  - Why? - Suppose we have 100 credit card transactions, of which 97 are legit and 3 are fraud and let's say we came up a model that predicts everything as fraud. Precision and Recall for the example is shown in the fig below.



- Now, if we simply take arithmetic mean of both, then it comes out to be nearly 51%. We shouldn't be giving such a moderate score to a terrible model since it's just predicting every transaction as fraud.
- So, we need something more balanced than the arithmetic mean and that is harmonic mean.
- The Harmonic mean is given by the formula shown in the figure on the bottom right.
- Harmonic mean is kind of an average when x and y are equal. But when x and y are different, then it's closer to the smaller number as compared to the larger number.
- For our previous example, F1 Score = Harmonic Mean(Precision, Recall)

  F1 Score = 2 * Precision * Recall / (Precision + Recall) = 2*3*100/103 = 5%
- So if one number is really small between precision and recall, the F1 Score kind of raises a flag and is more closer to the smaller number than the bigger one, giving the model an appropriate score rather than just an arithmetic mean.



- **Log Loss:**
  - Logarithmic Loss or Log Loss, works by penalising the false classifications.
  - It works well for multi-class classification. When working with Log Loss, the classifier must assign probability to each class for all the samples.
  - Suppose, there are N samples belonging to M classes, then the Log Loss is calculated as below :

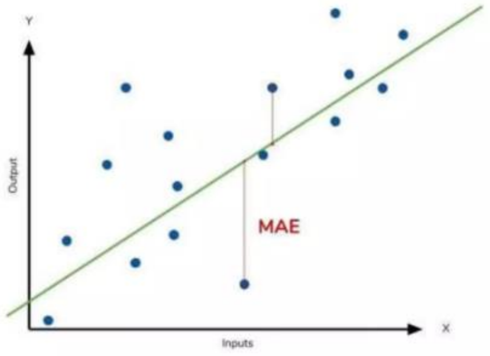
$$LogarithmicLoss = \frac{-1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} * \log(p_{ij})$$

    where,

    y_ij, indicates whether sample i belongs to class j or not

    p_ij, indicates the probability of sample i belonging to class j
  - Log Loss has no upper bound and it exists on the range [0, o0). Log Loss nearer to 0 indicates higher accuracy, whereas if the Log Loss is away from 0 then it indicates lower accuracy.
  - In general, minimising Log Loss gives greater accuracy for the classifier.
- **Mean Absolute Error:**
  - Mean Absolute Error is the average of the difference between the original values and the predicted values.
  - It gives us the measure of how far the predictions were from the actual output. However, they don't gives us any idea of the direction of the error i.e. whether we are under predicting the data or over predicting the data.
  - Mathematically, it is represented as :

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$



- **Mean Squared Error:**
  - Mean Squared Error(MSE) is quite similar to Mean Absolute Error, the only difference being that MSE takes the average of the square of the difference between the original values and the predicted values.
  - As, we take square of the error, the effect of larger errors become more pronounced then smaller error, hence the model can now focus more on the larger errors.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

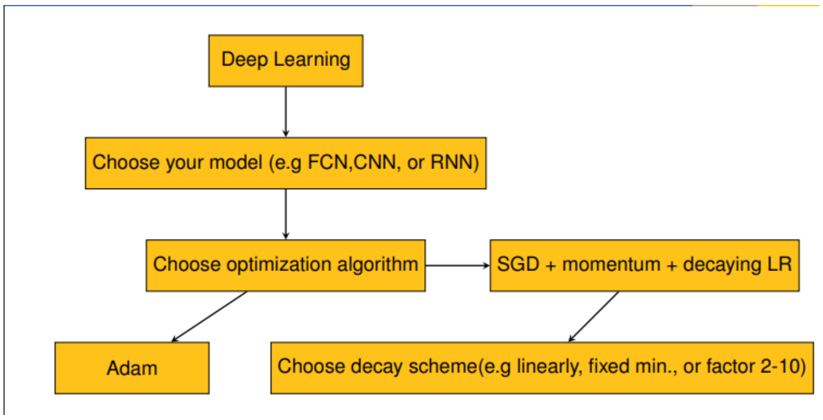
  - Instead of MSE, we generally use RMSE, which is equal to the square root of MSE.
  - Taking the square root of the average squared errors has some interesting implications for RMSE. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors.
  - This means the RMSE should be more useful when large errors are particularly undesirable.

Default Baseline Models:

- After selecting performance metrics and goals, the next important step in any deep learning application is to establish a reasonable baseline model. A baseline model serves as the starting point for experimentation and improvement.

- Depending on the complexity of the problem, sometimes it is better to start without deep learning. For simpler problems that can be solved by learning a few linear weights, simple models like logistic regression can be used.
- However, for complex “AI-complete” tasks such as **object recognition, speech recognition, or machine translation**, it is better to begin with appropriate **deep learning models**.
- The choice of model mainly depends on the structure of the data. For **supervised learning** with fixed-size vector inputs, a **feedforward neural network** with fully connected layers is suitable.
- For data with topological structures, such as **images**, **convolutional neural networks (CNNs)** are preferred. For **sequential data** like speech or text, **gated recurrent networks** such as **LSTM** or **GRU** are effective.
- Activation functions like **ReLU**, **Leaky ReLU**, or **Maxout** are commonly used because they improve learning performance.
- For optimization, **Stochastic Gradient Descent (SGD)** with **momentum** and a **decaying learning rate** is a standard choice. Other good alternatives include the **Adam optimizer**.
- Techniques like **batch normalization** can significantly improve both training speed and stability, especially for deep networks. If optimization becomes difficult, batch normalization should be added quickly.

- Regularization is essential to prevent overfitting, especially when the training dataset is not very large.
- Common regularization methods include **early stopping** and **dropout**, which are simple and effective. Batch normalization can sometimes also act as a regularizer, reducing the need for dropout.


Deep Learning
Choose your model (e.g FCN,CNN, or RNN)
Choose optimization algorithm
SGD + momentum + decaying LR
Adam
Choose decay scheme(e.g linearly, fixed min., or factor 2-10)

- If a similar problem has already been studied, it is useful to **copy the architecture or pretrained model** that performed well on that task.
- For example, convolutional networks trained on **ImageNet** are often reused for other vision tasks through **transfer learning**.
- In some domains, **unsupervised learning** can be added to the baseline. In **natural language processing**, techniques such as **word embeddings** learned from unlabeled data improve performance.
- In contrast, in **computer vision**, unsupervised learning is not always beneficial unless labeled data is limited. Therefore, unsupervised methods should be included only if they are known to help the specific task.
- the goal of a default baseline model is to build a reliable starting system that performs reasonably well and provides a reference point for further improvements in architecture, optimization, or regularization.

Determining Whether to Gather More Data:

- After establishing the first end-to-end system in a machine learning project, the next step is to measure its performance and determine whether to improve the algorithm or gather more data.
- In many cases, beginners attempt to enhance results by changing algorithms, but it is often more beneficial to collect additional data rather than modify the model.
- The first step in deciding whether to gather more data is to evaluate the **training performance**. If the performance on the training set is poor, it indicates that the model is not learning effectively from the available data. In such cases, gathering more data will not help.

- Instead, the model capacity should be increased by adding more layers or hidden units, or by improving the learning algorithm through tuning hyperparameters like the learning rate.
- If even large models with proper tuning do not perform well, the issue might lie in the **quality of the data**, which could be noisy, incomplete, or missing essential features. In that case, cleaner or richer data should be collected.
- If the **training performance is good**, the next step is to check the **test performance**. If both training and test performances are acceptable, no further action is needed.
- However, if the test performance is much worse than the training performance, it indicates **overfitting**. One of the most effective solutions in this case is to gather more data, as it helps improve generalization and reduce the gap between training and test errors.
- The decision depends on the **cost, feasibility, and expected benefit** of collecting more data. For example, in large-scale internet applications, gathering massive datasets is affordable and often the best solution.
- In contrast, in medical or scientific domains, collecting more data may be expensive or infeasible.
- When collecting additional data is not practical, **reducing model complexity** or **increasing regularization** (using techniques such as dropout or adjusting weight decay coefficients) can help control overfitting. If these methods fail to reduce the train-test gap, then gathering more data becomes necessary.
- To decide **how much data to collect**, one can plot a **learning curve** showing the relationship between training set size and generalization error. Extrapolating this curve helps estimate how much more data is needed to achieve a specific performance level.
- Generally, adding a small fraction of new data may not significantly impact generalization, so it is recommended to increase data size gradually on a **logarithmic scale**, such as doubling the number of samples in each experiment.
- gathering more data is useful when the model generalizes poorly despite good training performance and sufficient regularization. However, when the model underfits, has poor training accuracy, or data collection is infeasible, improving the model or data quality should be prioritized.

Selecting Hyperparameters:

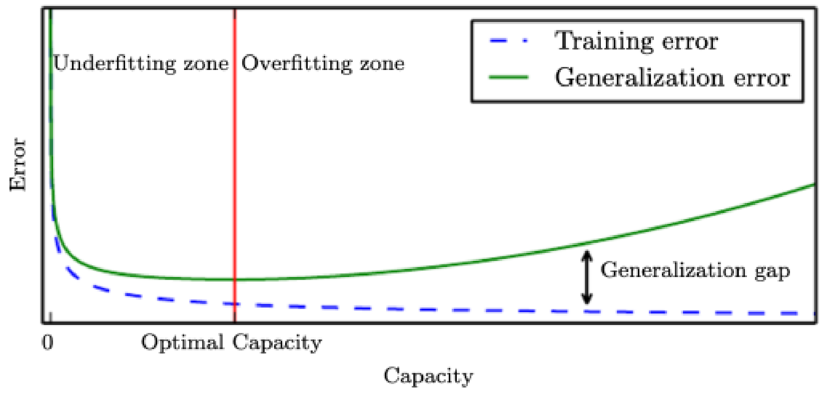
- Most deep learning algorithms come with many hyperparameters that control many aspects of the algorithm’s behavior. Some of these hyperparameters affect the time and memory cost of running the algorithm.
- There are two basic approaches to choosing these hyperparameters: choosing them manually and choosing them automatically.
- Choosing the hyperparameters manually requires understanding what the hyperparameters do and how machine learning models achieve good generalization.
- Automatic hyperparameter selection algorithms greatly reduce the need to understand these ideas, but they are often much more computationally costly.
- Hyperparameters control different aspects of how your model behaves.

  1.Costs: time and memory requirements during training (and inference)

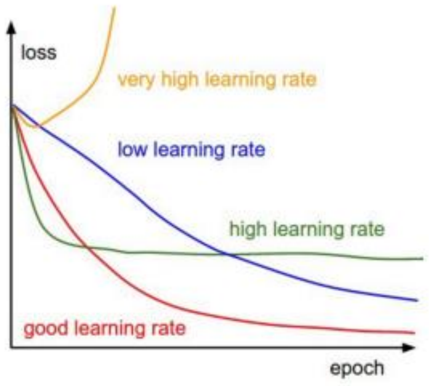
  2.Quality: performance during training process and on new inputs

- Goal :

  Find hyperparameters that minimize the generalization error, s.t. they do not exceed our runtime and memory requirements.
- **Manual Hyperparameter Tuning:**
  - We want the capacity of our model to match the complexity of our task.
  - The effective model capacity consists of three factors:
    - Representational capacity
    - Cost function capacity
    - Regularization capacity
  - The goal of manual hyperparameter search is usually to find the lowest general ization error subject to some runtime and memory budget.
  - The primary goal of manual hyperparameter search is to adjust the effective capacity of the model to match the complexity of the task.
  - The generalization error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters, as in figure.



- A model with more layers and more hidden units per layer has higher representational capacity—it is capable of representing more complicated functions.
- Many hyperparameters are discrete, such as the number of units in a layer or the number of linear pieces in a max out unit, so it is only possible to visit a few points along the curve. Some hyperparameters are binary.



The learning rate is a very important hyperparameter.

- too large: gradient descent can increase training error
- too low: slower training, more likely to get stuck in local minima

The model capacity is highest if the learning rate is set correctly!

- It controls the effective capacity of the model in a more complicated way than other hyperparameters—the effective capacity of the model is highest when the learning rate is correct for the optimization problem, not when the learning rate is especially large or especially small.
- The learning rate has a U-shaped curve for training error, illustrated in figure 11.1 .
- Tuning the parameters other than the learning rate requires monitoring both training and test error to diagnose whether your model is overfitting or underfitting, then adjusting its capacity appropriately.

- If your error on the training set is higher than your target error rate, you have no choice but to increase capacity.

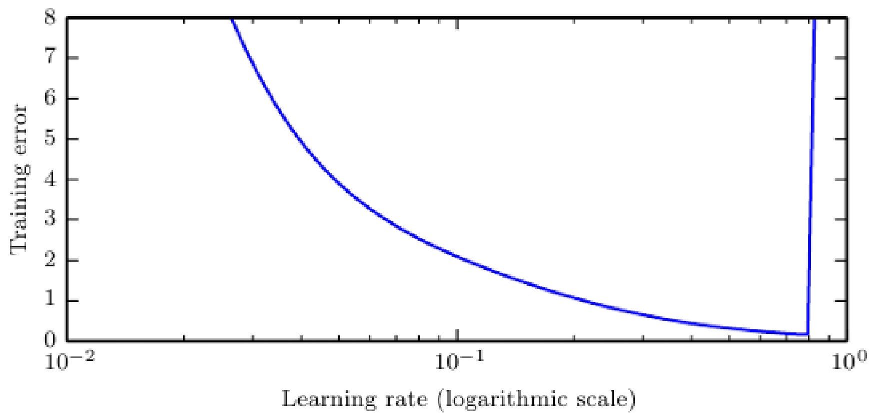
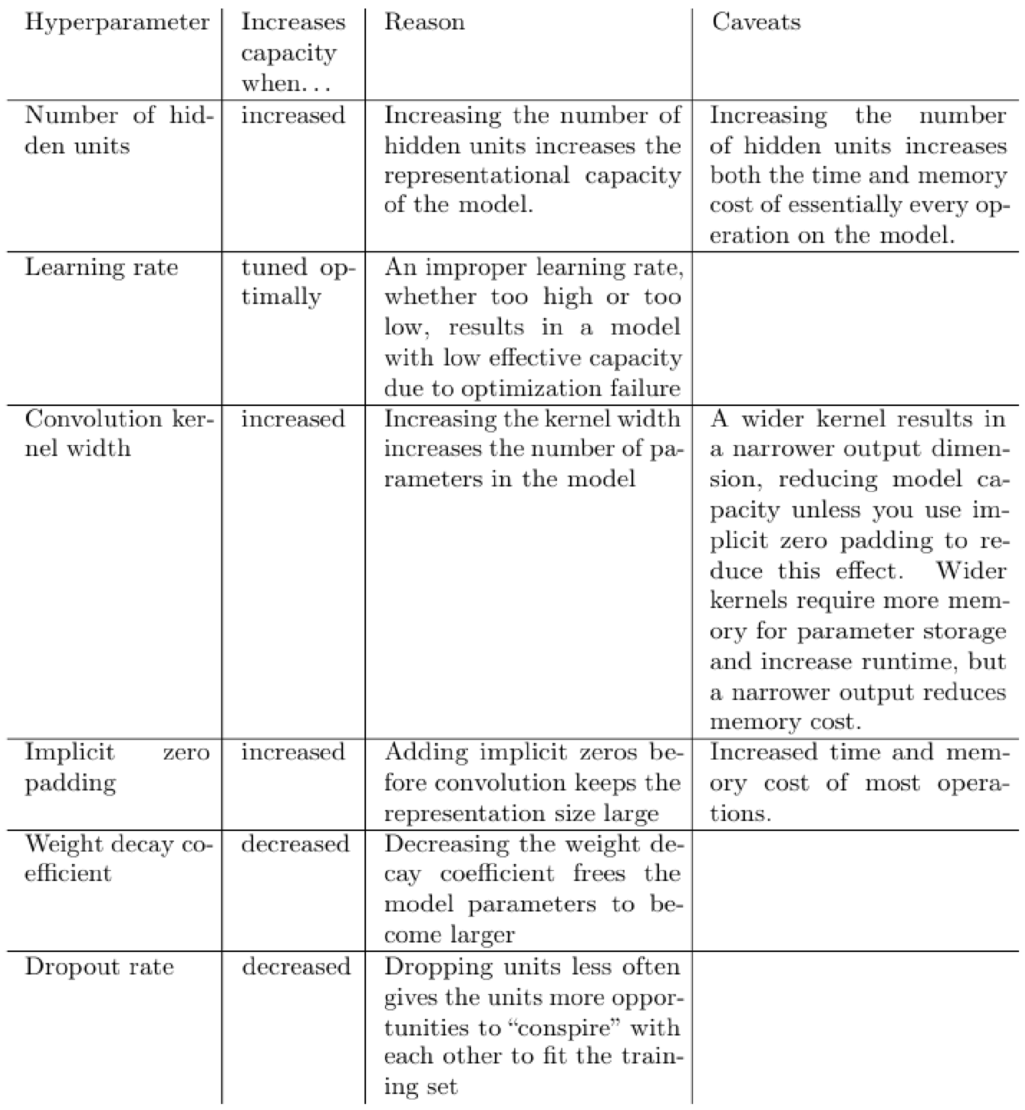


Figure 11.1: Typical relationship between the learning rate and the training error.

- The brute force way to practically guarantee success is to continually increase model capacity and training set size until the task is solved.
- The optimal test error is found by trading off these quantities. Neural networks typically perform best when the training error is very low (and thus, when capacity is high) and the test error is primarily driven by the gap between train and test error.
- While manually tuning hyperparameters, do not lose sight of your end goal: good performance on the test set. Adding regularization is only one way to achieve this goal.
- This approach does of course increase the computational cost of training and inference, so it is only feasible given appropriate resources.
- Most hyperparameters can be set by reasoning about whether they increase or decrease model capacity. Some examples are included in Table 11.1 .



Hyperparameter	Increases capacity when...	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure	
Convolution kernel width	increased	Increasing the kernel width increases the number of parameters in the model	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost.
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large	Increased time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

Table 11.1: The effect of various hyperparameters on model capacity.

- **Automatic Hyperparameter Optimization Algorithms:**
  - An ideal learning algorithm should take a dataset and automatically output a good function without requiring manual hyperparameter tuning.
  - Algorithms like logistic regression and SVMs are popular because they perform well with only one or two hyperparameters to adjust.
  - Neural networks, however, often require tuning of many hyperparameters, sometimes forty or more, to achieve the best performance.
  - Manual tuning can be effective when users have prior experience or good starting points from similar tasks or architectures.
  - In many new applications, such reference points are unavailable, making automated hyperparameter optimization necessary.

  - Hyperparameter tuning can be viewed as an optimization process that seeks to minimize validation error within certain constraints such as training time or memory usage.
  - This leads to the idea of creating optimization algorithms that automatically select hyperparameters for learning algorithms.
  - These optimization algorithms, however, have their own hyperparameters, known as secondary hyperparameters.
  - Fortunately, these secondary hyperparameters are much easier to set and can often work well across different tasks.

- **Grid Search:**

  - When there are three or fewer hyperparameters, grid search is commonly used to find the best combination of values.
  - In this method, the user selects a finite set of possible values for each hyperparameter, and models are trained for every possible combination of these values.
  - The combination that gives the lowest validation set error is considered the best.
  - The range of values for each hyperparameter is usually chosen based on prior experience to ensure that the optimal value is likely included.

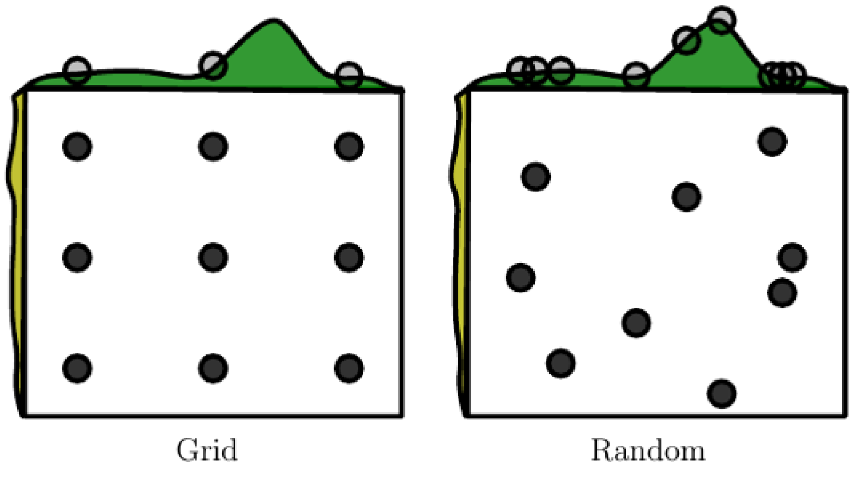


Figure : Comparison of grid search and random search.

  - For numerical hyperparameters, values are often picked on a logarithmic scale, such as learning rates from {0.1, 0.01, 0.001, 0.0001, 0.00001}.

- Grid search can be improved by performing it repeatedly—if the best value is at the edge of the current range, the search range is shifted or expanded; if it lies in the middle, a finer grid is applied to zoom in.
- This iterative process helps refine the hyperparameter selection more accurately.
- However, a major drawback of grid search is that its computational cost increases exponentially with the number of hyperparameters.
- If there are m hyperparameters, each with n possible values, the total number of required experiments grows as O(n^m).
- Although grid search can be parallelized to some extent since trials are independent, the exponential cost still makes it inefficient when dealing with many hyperparameters.

- **Random Search:**

  - there is an alternative to grid search that is as simple to program, more convenient to use, and converges much faster to good values of the hyperparameters: random search.
  - A random search proceeds as follows. First we define a marginal distribution for each hyperparameter, e.g., a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters.
  - For example,

$$\texttt{log\_learning\_rate} \sim u(-1,-5)$$
$$\texttt{learning\_rate} = 10^{\text{log\_learning\_rate}}.$$

where u(a,b) indicates a sample of the uniform distribution in the interval (a,b). Similarly the log number of hidden units _ _ _ maybesampledfromu(log(50), log(2000)).

  - Unlike in the case of a grid search, one should not discretize or bin the values of the hyperparameters.
  - This allows one to explore a larger set of values, and does not incur additional computational cost.
  - As with grid search, one may often want to run repeated versions of random search, to refine the search based on the results of the first run.

- The main reason why random search finds good solutions faster than grid search is that there are no wasted experimental runs, unlike in the case of grid search, when two values of a hyperparameter (given values of the other hyperparameters) would give the same result.
- In the case of grid search, the other hyperparameters would have the same values for these two runs, whereas with random search, they would usually have different values.
- Hence if the change between these two values does not marginally make much difference in terms of validation set error, grid search will unnecessarily repeat two equivalent experiments while random search will still give two independent explorations of the other hyperparameters.

- **Model-Based Hyperparameter Optimization:**

  - The search for good hyperparameters can be cast as an optimization problem. The decision variables are the hyperparameters.
  - The cost to be optimized is the validation set error that results from training using these hyperparameters. In simplified settings where it is feasible to compute the gradient of some differentiable error measure on the validation set with respect to the hyperparameters, we can simply follow this gradient.
  - To compensate for this lack of a gradient, we can build a model of the validation set error, then propose new hyperparameter guesses by performing optimization within this model.
  - Most model-based algorithms for hyperparameter search use a Bayesian regression model to estimate both the expected value of the validation set error for each hyperparameter and the uncertainty around this expectation.
  - Optimization thus involves a trade off between exploration (proposing hyperparameters for which there is high uncertainty, which may lead to a large improvement but may also perform poorly) and exploitation (proposing hyperparameters which the model is confident will perform as well as any hyperparameters it has seen so far—usually hyperparameters that are very similar to ones it has seen before).
  - Contemporary approaches to hyperparameter optimization include Spearmint ,TPE and SMAC .

  - Currently, we cannot unambiguously recommend Bayesian hyperparameter optimization as an established tool for achieving better deep learning results or for obtaining those results with less effort.
  - One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from the experiment.
  - At various time points, the hyperparameter optimization algorithm can choose to begin a new experiment, to “freeze” a running experiment that is not promising, or to “thaw” and resume an experiment that was earlier frozen but now appears promising given more information.

<u>Debugging Strategies:</u>

- When a machine learning system performs poorly, it is usually difficult to tell whether the poor performance is intrinsic to the algorithm itself or whether there is a bug in the implementation of the algorithm.
- In most cases, we do not know a priori what the intended behavior of the algorithm is.
- If we train a neural network on a new classification task and it achieves 5% test error, we have no straightforward way of knowing if this is the expected behavior or sub-optimal behavior.
- If one part is broken, the other parts can adapt and still achieve roughly acceptable performance.
- For example, suppose that we are training a neural net with several layers parametrized by weights W and biases b. Suppose further that we have manually implemented the gradient descent rule for each parameter separately, and we made an error in the update for the biases:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha$$

- where α is the learning rate. This erroneous update does not use the gradient at all. It causes the biases to constantly become negative throughout learning, which is clearly not a correct implementation of any reasonable learning algorithm.
- Most debugging strategies for neural nets are designed to get around one or both of these two difficulties.

- Some important debugging tests include:
  - Visualize the model in action:
    - When training a model to detect objects in images, view some images with the detections proposed by the model displayed superimposed on the image.
    - When training a generative model of speech, listen to some of the speech samples it produces.
    - This may seem obvious, but it is easy to fall into the practice of only looking at quantitative performance measurements like accuracy or log-likelihood.
    - Evaluation bugs can be some of the most devastating bugs because they can mislead you into believing your system is performing well when it is not.
  - Visualize the worst mistakes:
    - Most models are able to output some sort of confidence measure for the task they perform.
    - The probability assigned to the most likely class thus gives an estimate of the confidence the model has in its classification decision.
    - By viewing the training set examples that are the hardest to model correctly, one can often discover problems with the way the data has been preprocessed or labeled.
    - Modifying the detection system to crop much wider images resulted in much better performance of the overall system, even though the transcription network needed to be able to process greater variation in the position and scale of the address numbers.
  - Reasoning about software using train and test error:
    - It is often difficult to determine whether the underlying software is correctly implemented. Some clues can be obtained from the train and test error.
    - If training error is low but test error is high, then it is likely that that the training procedure works correctly, and the model is overfitting for fundamental algorithmic reasons.

- An alternative possibility is that the test error is measured incorrectly due to a problem with saving the model after training then reloading it for test set evaluation, or if the test data was prepared differently from the training data.
- If both train and test error are high, then it is difficult to determine whether there is a software defect or whether the model is underfitting due to fundamental algorithmic reasons.

➢ Fit a tiny dataset:

- If you have high error on the training set, determine whether it is due to genuine underfitting or due to a software defect.
- Usually even small models can be guaranteed to be able fit a sufficiently small dataset.
- For example, a classification dataset with only one example can be fit just by setting the biases of the output layer correctly.
- Usually if you cannot train a classifier to correctly label a single example, an autoencoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is a software defect preventing successful optimization on the training set.
- This test can be extended to a small dataset with few examples.

➢ Compare back-propagated derivatives to numerical derivatives:

- If you are using a software framework that requires you to implement your own gradient computations, or if you are adding a new operation to a differentiation library and must define its b prop method, then a common source of error is implementing this gradient expression incorrectly.
- One way to verify that these derivatives are correct is to is to compare the derivatives computed by your implementation of automatic differentiation to the derivatives computed by a finite differences. Because ,

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon},$$

we can approximate the derivative by using a small, finite ϵ:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

We can improve the accuracy of the approximation by using the centered difference:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}.$$

- The perturbation size ϵ must chosen to be large enough to ensure that the perturbation is not rounded down too much by finite-precision numerical computations.
- Usually, we will want to test the gradient or Jacobian of a vector-valued function g : Rm →Rn.
- We can either run finite differencing m n times to evaluate all of the partial derivatives of g, or we can apply the test to a new function that uses random projections at both the input and output of g.

➢ Monitor histograms of activations and gradient:

- It is often useful to visualize statistics of neural network activations and gradients, collected over a large amount of training iterations (maybe one epoch).

- The pre-activation value of hidden units can tell us if the units saturate, or how often they do.
- Finally, it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves.
- As suggested by Bottou 2015 ( ), we would like the magnitude of parameter updates over a minibatch to represent something like 1% of the magnitude of the parameter, not 50% or 0.001% (which would make the parameters move too slowly).
- It may be that some groups of parameters are moving at a good pace while others are stalled. When the data is sparse (like in natural language), some parameters may be very

rarely updated, and this should be kept in mind when monitoring their evolution.

- Typically these can be debugged by testing each of their guarantees. Some guarantees that some optimization algorithms offer include that the objective function will never increase after one step of the algorithm, that the gradient with respect to some subset of variables will be zero after each step of the algorithm, and that the gradient with respect to all variables will be zero at convergence.

Example: Multi-Digit Number Recognition:

- To provide an end-to-end description of how to apply our design methodology in practice, we present a brief account of the Street View transcription system, from the point of view of designing the deep learning components.
- Obviously, many other components of the complete system, such as the Street View cars, the database infrastructure, and so on, were of paramount importance.
- From the point of view of the machine learning task, the process began with data collection. The cars collected the raw data and human operators provided labels.
- **Goal**

  Assign digits to pictures of street numbers if model confidence p(y| x) ≥ t for some threshold t.

Figure: Street view transcription system.

- Steps:
  - Choose performance metrics:
    - Choose the metrics according to the project's business goal!
    - Here: maps require high, human-level accuracy
    - This meant a high threshold to accept results of the model

    Metric is therefore: coverage, i.e. percentage of confidences above the threshold (goal: >95%)

  - Establish baseline model
    - Try to iteratively improve the model!
    - Here: n different softmax units to predict n characters
    - each unit trained independently
    - p(y|x) obtained by multiplying units together

    Improvement idea: use output layer/cost function that computes log-likelihood instead

    Coverage was still below 90%. Is the problem under- or overfitting?

  - Debug model
    - Here: training and test error were identical

      → underfitting or problem with training data

    - Visualize model's worst mistakes

      → Some images were cropped too tightly!

    Solution: add margin of safety around crops (+10% coverage)

  - Adjust hyperparameters
    - Here: train and test error remained equal → underfitting
    - Model was made larger

- Overall, the transcription project was a great success, and allowed hundreds of millions of addresses to be transcribed both faster and at lower cost than would have been possible via human effort.

Applications:

Large-Scale Deep Learning:

- Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior.
- It truly is important to emphasize the fact that the number of neurons must be large. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use.
- Because the size of neural networks is of paramount importance, deep learning requires high performance hardware and software infrastructure.

  - **Fast CPU Implementations:**

    - Earlier, neural networks were trained using the CPU of a single machine, but this approach is now considered inadequate due to high computational demands.
    - Modern neural network training typically relies on GPUs or multiple networked CPUs for faster processing. Before adopting these advanced setups, researchers proved that CPUs alone could not handle the heavy workloads efficiently.
    - However, careful CPU-specific optimization can still provide significant speed improvements.
    - For instance, in 2011, using fixed-point arithmetic instead of floating-point arithmetic gave up to a threefold speedup in neural network computations.
    - The performance advantage depends on the CPU model, as some may favor floating-point operations instead.

- Efficient numerical computation often requires optimizing data structures to reduce cache misses and leveraging vectorized instructions.
- Many researchers overlook such low-level optimizations, but poor implementation efficiency can limit model size and, in turn, reduce its accuracy.

➢ **GPU Implementations:**

- Modern neural network implementations mainly rely on GPUs, which were originally developed for rendering graphics in video games.
- The parallel computation and high memory bandwidth required for graphics processing also make GPUs ideal for neural network training.
- GPUs can handle large matrix operations and process multiple data points simultaneously, providing significant speed advantages over CPUs.
- Early research in the mid-2000s showed that implementing neural networks on GPUs led to several-fold speedups compared to CPU-based systems.
- The development of general-purpose GPUs (GP-GPUs) and programming frameworks like NVIDIA’s CUDA made it easier to write custom code for deep learning, accelerating their widespread adoption.
- However, writing efficient GPU code is complex, as it requires managing memory access, thread coordination, and minimizing branching for optimal performance.
- To simplify this, researchers use optimized libraries such as Theano, cuda-convnet, TensorFlow, and Torch, which provide high-performance GPU operations without needing to write low-level GPU code manually.

➢ **Large-Scale Distributed Implementations:**

- In many cases, the computational resources available on a single machine are insufficient. We therefore want to distribute the workload of training and inference across many machines.

- Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as data parallelism .
- It is also possible to get model parallelism, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training.
- Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD step, but usually we get less than linear returns in terms of optimization performance.
- This can be solved using asynchronous stochastic gradient descent
- pioneered the multi-machine implementation of this lock-free approach to gradient descent, where the parameters are managed by a parameter server rather than stored in shared memory.
- Academic deep learning researchers typically cannot afford the same scale of distributed learning systems but some research has focused on how to build distributed networks with relatively low-cost hardware available in the university setting.

➢ **Model Compression:**

- In many commercial applications, minimizing the time and memory cost of inference is more important than reducing training costs.
- Models are often trained once using powerful machines and then deployed to billions of users, many of whom have limited computational resources, such as mobile devices.
- A key method for reducing inference cost is model compression, which replaces a large, expensive model with a smaller one that requires less memory and computation.
- Model compression is useful when large models are needed mainly to avoid overfitting or when ensembles of models improve generalization but are too costly to evaluate.
- Once a large model has learned a function (f(x)), it can generate unlimited synthetic training data by applying (f) to sampled inputs, allowing a smaller model to be trained to mimic its outputs.

- Sampling new inputs similar to real test data helps the smaller model learn efficiently, which can be achieved by corrupting training data or using a generative model.
- Alternatively, the smaller model can be trained directly on the original data to reproduce not just the predictions but also additional information, such as the larger model's posterior distribution over incorrect classes.

➢ **Dynamic Structure:**

- One strategy for accelerating neural network computations is to use dynamic structure, where only a subset of networks or features is computed based on the input.
- Conditional computation allows networks to run only the relevant parts for each input, reducing unnecessary calculations.
- Cascades of classifiers can speed up inference by using low-capacity models to filter out easy cases and high-capacity models only when needed, achieving high precision efficiently.
- Dynamic routing can also be implemented using mixture of experts, where a gating network selects which expert or subset of units to activate, potentially reducing computational cost.
- Hard switches and attention mechanisms offer dynamic input selection, though fully exploiting computational benefits remains challenging at large scale.
- A major challenge with dynamically structured systems is reduced parallelism, making CPU and GPU implementations less efficient due to branching and memory access patterns.
- Strategies like grouping inputs that follow the same branch can partially mitigate these issues, though real-time processing may still face load-balancing challenges.

➢ **Specialized Hardware Implementations of Deep Networks:**

- Since the early days of neural networks, specialized hardware has been developed to accelerate training and inference, including ASICs, digital, analog, hybrid, and more recently, FPGA implementations.

- While general-purpose CPUs and GPUs typically use 32 or 64-bit floating-point representations, lower-precision formats have been shown to be sufficient, especially for inference.
- The growing popularity of deep learning and the slower improvements in single-core CPU or GPU performance have increased interest in specialized hardware.
- Low-precision implementations, using between 8 and 16 bits, can support backpropagation-based neural networks, with training generally requiring higher precision than inference.
- Dynamic fixed-point representations allow a shared range among groups of numbers, reducing the number of bits needed while maintaining accuracy.
- Using lower-precision fixed-point numbers reduces hardware area, power consumption, and computation time, particularly for multiplications, which are the most demanding operations in deep network training and usage.

Computer Vision:

- Computer vision has been a major focus of deep learning because it is easy for humans but challenging for computers, with popular benchmarks including object recognition and optical character recognition.
- Applications range from replicating human visual abilities, like face recognition, to novel tasks such as inferring sound from visible vibrations in video.
- Most deep learning research in computer vision focuses on object recognition, detection, annotation, and pixel-level labeling rather than exotic applications.
- Additionally, deep learning is widely used for image synthesis and restoration, which, while not strictly computer vision, supports tasks like repairing or modifying images.

Preprocessing:

- Many deep learning applications require sophisticated preprocessing, but computer vision typically needs minimal preprocessing beyond standardizing pixel values to a consistent range, such as [0,1] or [-1,1].

- Images often need to be cropped or scaled to a standard size, although some convolutional models can handle variable-sized inputs or outputs.
- Dataset augmentation is a common preprocessing technique for training, improving generalization by creating varied versions of inputs, and can also be applied at test time as an ensemble approach.
- Additional preprocessing may reduce irrelevant variability in the data to simplify the task, lower generalization error, and allow smaller models to perform well, though with large datasets and models, such preprocessing is often unnecessary.

❖ **Contrast Normalization:**

- One of the most obvious sources of variation that can be safely removed for many tasks is the amount of contrast in the image. Contrast simply refers to the magnitude of the difference between the bright and the dark pixels in an image.
- There are many ways of quantifying the contrast of an image. In the context of deep learning, contrast usually refers to the standard deviation of the pixels in an image or region of an image.
- Suppose we have an image represented by a tensor X∈R*r*c*3, with X i,j,1 being the red intensity at row i and column j, X i,j,2 giving the green intensity and X i,j,3 giving the blue intensity. Then the contrast of the entire image is given by

$$\sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}\left(X_{i,j,k}-\bar{\mathbf{X}}\right)^2}$$

where X is the mean intensity of the entire image:

$$\bar{\mathbf{X}} = \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k}.$$

- Global contrast normalization (GCN) aims to prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant s.
- Given an input image X, GCN produces an output image X⍰, defined such that

$$X'_{i,j,k} = s \frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}\left(X_{i,j,k} - \bar{X}\right)^2}\right\}}.$$

- It is preferable to define GCN in terms of standard deviation rather than L2 norm because the standard deviation includes division by the number of pixels, so GCN based on standard deviation allows the same s to be used regardless of image size.
- there is a preprocessing operation known as sphering and it is not the same operation as GCN.

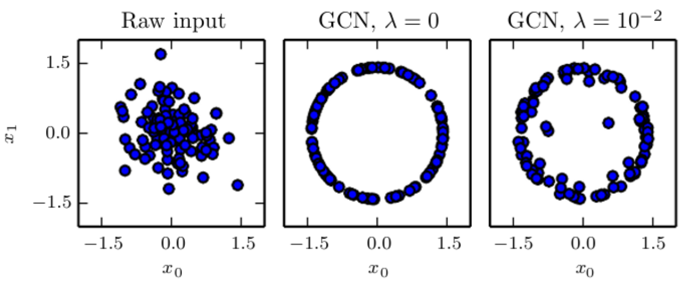


Figure : GCN maps examples onto a sphere.

- Sphering does not refer to making the data lie on a spherical shell, but rather to rescaling the principal components to have equal variance, so that the multivariate normal distribution used by PCA has spherical contours. Sphering is more commonly known as whitening .

- Global contrast normalization will often fail to highlight image features we would like to stand out, such as edges and corners.
- This motivates local contrast normalization. Local contrast normalization ensures that the contrast is normalized across each small window, rather than over the image as a whole.
- Various definitions of local contrast normalization are possible. In all cases, one modifies each pixel by subtracting a mean of nearby pixels and dividing by a standard deviation of nearby pixels.
- In the case of color images, some strategies process different color channels separately while others combine information from different channels to normalize each pixel.
- Local contrast normalization is a differentiable operation and can also be used as a nonlinearity applied to the hidden layers of a network, as well as a preprocessing operation applied to the input.

❖ **Dataset Augmentation:**

- it is easy to improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class.
- Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations.
- As described before, classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset.

  - In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation.

Speech Recognition:

- The task of speech recognition is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker.
- Let X = (x(1),x(2),...,x( ) T) denote the sequence of acoustic input vectors (traditionally produced by splitting the audio into 20ms frames).
- Let y = (y1,y2,...,yN) denote the target output sequence (usually a sequence of words or characters).
- The automatic speech recognition (ASR) task consists of creating a function f∗ ASR that computes the most probable linguistic sequence given the acoustic sequence :

$$f^*_{\mathrm{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X})$$

  where P∗ is the true conditional distribution relating the inputs X to the targets y.

- From the 1980s until around 2009–2012, state-of-the-art speech recognition primarily relied on GMM-HMM systems, with GMMs modeling the association between acoustic features and phonemes and HMMs modeling phoneme sequences.
- Early neural network-based ASR systems in the late 1980s and early 1990s matched or slightly exceeded GMM-HMM performance, but industry continued using GMM-HMMs due to engineering investment.
- Starting in 2009, deep learning approaches using unsupervised pretraining with restricted Boltzmann machines (RBMs) built deep feedforward networks that significantly improved phoneme recognition, reducing error rates on TIMIT from about 26% to 20.7%. Extensions included speaker-adaptive features and large-vocabulary recognition, while later deep networks used rectified linear units and dropout, making unsupervised pretraining less necessary.
- These advances led to around a 30% improvement in word error rates, prompting rapid adoption of deep learning in industrial ASR products.

- Innovations such as two-dimensional convolutional networks improved modeling of spectrograms by replicating weights across time and frequency, and end-to-end deep learning systems using deep LSTM RNNs and the CTC framework further reduced phoneme error rates to 17.7%, allowing models to learn acoustic-to-phonetic alignment directly.
- Deep learning in speech recognition shifted from using neural networks for feature extraction in GMM-HMM systems to fully replacing GMMs for mapping acoustic features to phonemes or sub-phonemic states.
- Large labeled datasets and deeper network architectures enabled dramatic improvements in recognition accuracy compared to earlier models.
- Convolutional networks in ASR improved over time-delay neural networks by treating spectrograms as images with both time and frequency axes, allowing better weight sharing and feature extraction.
- End-to-end deep learning approaches eliminated the need for HMMs, enabling models like deep LSTM RNNs to capture both temporal and hierarchical depth in speech sequences.
- These advances not only lowered phoneme error rates but also enabled real-world deployment in commercial products, such as mobile phones, significantly transforming the ASR industry.

Natural Language Processing:

- Natural language processing (NLP) involves enabling computers to understand and generate human languages, which are often ambiguous and difficult to formally describe.
- NLP applications include tasks like machine translation, where a sentence in one language must be converted into an equivalent sentence in another language.
- Many NLP systems rely on language models that define probability distributions over sequences of words, characters, or bytes.
- Generic neural network techniques can be applied successfully to NLP, but achieving high performance and scalability often requires domain-specific strategies.
- Efficient NLP models usually treat language as a sequence of words rather than individual characters or bytes.
- Word-based models must handle extremely high-dimensional and sparse discrete spaces, and specialized computational and statistical techniques are used to manage this complexity effectively.

1.n-grams:

- A language model defines a probability distribution over sequences of tokens in a natural language.
- Depending on how the model is designed, a token may be a word, a character, or even a byte. Tokens are always discrete entities.
- The earliest successful language models were based on models of fixed-length sequences n n of tokens called-grams. An-gram is a sequence of tokens.
- Models based on n-grams define the conditional probability of the n-th token given the preceding n − 1 tokens.
- The model uses products of these conditional distributions to define the probability distribution over longer sequences:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}).$$

- This decomposition is justified by the chain rule of probability. The probability distribution over the initial sequenceP(x1,...,xn−1) may be modeled by a different model with a smaller value of .
- Training n-gram models is straightforward because the maximum likelihood estimate can be computed simply by counting how many times each possible n gram occurs in the training set.
- For small values of n, models have particular names: unigram These names derive from the Latin prefixes for the corresponding numbers and the Greek suffix "-gram" denoting something that is written. for n=1, bigram for n=2, and trigram for n=3.
- Usually we train both an n-gram model and ann−1 grammodel simultaneously. This makes it easy to compute simply by looking up two stored probabilities.

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \ldots, x_t)}{P_{n-1}(x_{t-n+1}, \ldots, x_{t-1})}$$

- As an example, we demonstrate how a trigram model computes the probability of the sentence "THE DOG RAN AWAY."

- The first words of the sentence cannot be handled by the default formula based on conditional probability because there is no context at the beginning of the sentence.
- Instead, we must use the marginal prob ability over words at the start of the sentence.
- We thus evaluate P3(THE DOG RAN). Finally, the last word may be predicted using the typical case, of using the conditional distribution P (AWAY| DOG RAN).
- we obtain:

$$P(\texttt{THE DOG RAN AWAY}) = P_3(\texttt{THE DOG RAN})P_3(\texttt{DOG RAN AWAY})/P_2(\texttt{DOG RAN}).$$

- A fundamental limitation of maximum likelihood for n-gram models is that Pn as estimated from training set counts is very likely to be zero in many cases, even though the tuple (xt n − +1,...,xt) may appear in the test set.
- To avoid such catastrophic outcomes, most n-gram models employ some form of smoothing.
- Back-off methods look-up the lower-order n-grams if the frequency of the context xt−1,...,xt n − +1 is too small to use the higher-order model.
- To improve the statistical efficiency of n-gram models, class-based language introduce the notion of word categories and then share statistical strength between words that are in the same category.

2.Neural Language Models:

- Neural language models (NLMs) are designed to overcome the curse of dimensionality in modeling natural language by using distributed representations, or word embeddings, that allow the model to recognize similarities between words while keeping them distinct.
- These embeddings enable statistical sharing between words and their contexts, allowing the model to generalize from each training sentence to an exponential number of semantically related sentences.

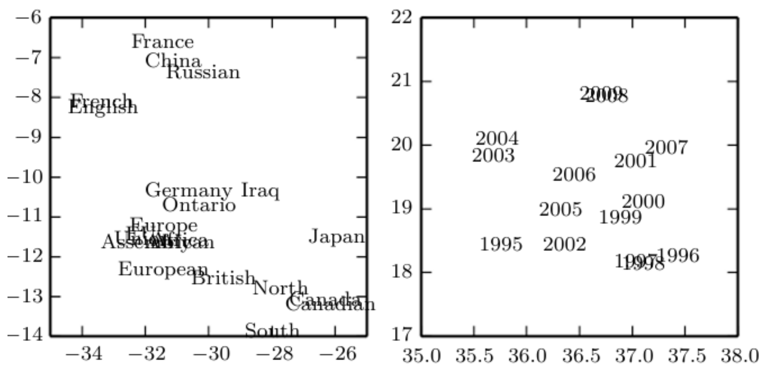


Figure: Two-dimensional visualizations

- Word embeddings map words from a high-dimensional one-hot space to a lower-dimensional feature space, placing words with similar meanings close together.
- This approach allows NLMs to capture semantic relationships and improve predictions across related contexts.
- The concept of embeddings is not limited to NLP; similar distributed representations are used in other domains, such as image embeddings in convolutional networks.
- Distributed representations can also be applied in graphical models using multiple latent variables, extending the idea beyond neural networks.

3.High-Dimensional Outputs:

- In many natural language applications, models often need to produce words as outputs, which can be computationally expensive for large vocabularies containing hundreds of thousands of words.
- Representing an output distribution over such a large vocabulary typically requires applying an affine transformation from the hidden representation to the output space, followed by a softmax function.
- The weight matrix for this transformation is very large, resulting in high memory usage and computational cost.
- The softmax normalization across all vocabulary words necessitates full matrix multiplication during both training and testing, rather than only computing the dot product for the correct output.
- This leads to high computational costs at training time for calculating likelihoods and gradients, and at test time for computing probabilities for all or selected words.

- Suppose that h is the top hidden layer used to predict the output probabilities ˆ y. If we parametrize the transformation from h to ˆy with learned weights W and learned biases b, then the affine softmax output layer performs the following computations:

$$
\begin{aligned}
a_i &= b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\}, \\
\hat{y}_i &= \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}.
\end{aligned}
$$

- <u>Use of a Short List:</u>

    - The first neural language models dealt with the high cost of using a softmax over a large number of output words by limiting the vocabulary size to 10,000 or 20,000 words.
    - This may be achieved by adding an extra sigmoid output unit to provide an estimate of P(i C ∈ |T).
    - The extra output can then be used to achieve an estimate of the probability distribution over all words in as follows:

$$
\begin{aligned}
P(y = i \mid C) =& 1_{i\in\mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) \\
&+ 1_{i\in\mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C)
\end{aligned}
$$

    - With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.
    - This disadvantage has stimulated the exploration of alternative methods to deal with high-dimensional outputs, described below.

- <u>Hierarchical Softmax:</u>

    - A classical approach to reducing the computational burden of high-dimensional output layers over large vocabulary sets V is to decompose probabilities hierarchically.
    - Instead of necessitating a number of computations proportional to | | V (and also proportional to the number of hidden units, nh), the | | V factor can be reduced to as low as log| | V.

- This is typically done using a standard cross-entropy loss, corresponding to maximizing the log-likelihood of the correct sequence of decisions.
- Tools from information theory specify how to choose the optimal binary code given the relative frequencies of the words.

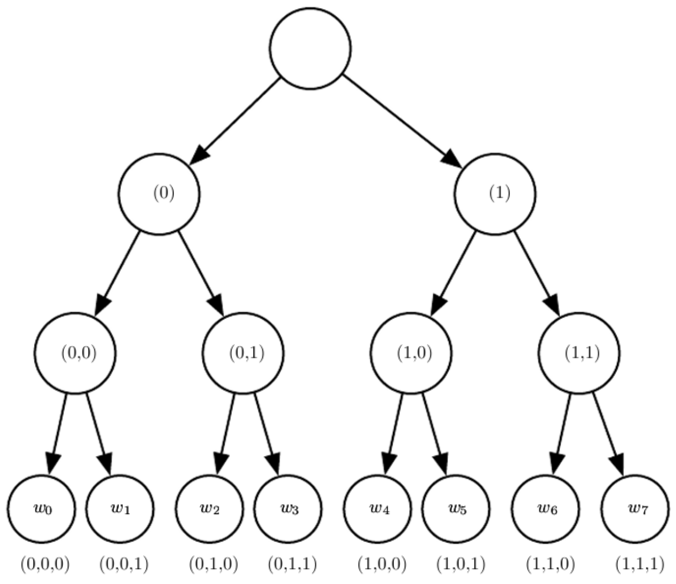


Figure : Illustration of a simple hierarchy of word categories

- The leaves of the tree represent actual specific words.
- Internal nodes represent groups of words.
- Any node can be indexed by the sequence of binary decisions (0=left, 1=right) to reach the node from the root.
- Super-class (0) contains the classes (0, 0) (0 and ,1), which respectively contain the sets of words{w0,w1} and {w2,w3}, and similarly super-class (1) contains the classes (1,0) (1 and ,1), which respectively contain the words (w4,w5) (and w6,w7).

- If the tree is sufficiently balanced, the maximum depth (number of binary decisions) is on the order of the logarithm of the number of words| | V: the choice of one out of | | V words can be obtained by doing O(log| | V)operations (one for each of the nodes on the path from the root).

- An important advantage of the hierarchical softmax is that it brings computa tional benefits both at training time and at test time, if at test time we want to compute the probability of specific words.

- A disadvantage is that in practice the hierarchical softmax tends to give worse test results than sampling-based methods we will describe next. This may be due to a poor choice of word classes.

- <u>Importance Sampling:</u>
  - One way to speed up the training of neural language models is to avoid explicitly computing the contribution of the gradient from all of the words that do not appear in the next position.
  - Using the notation introduced in equation , the gradient can be written as follows:

$$
\begin{aligned}
\frac{\partial \log P(y \mid C)}{\partial \theta} &= \frac{\partial \log \operatorname{softmax}_y(\boldsymbol{a})}{\partial \theta} \\
&= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \\
&= \frac{\partial}{\partial \theta}(a_y - \log \sum_i e^{a_i}) \\
&= \frac{\partial a_y}{\partial \theta} - \sum_i P(y = i \mid C) \frac{\partial a_i}{\partial \theta}
\end{aligned}
$$

  - The first term is the positive phase term (pushing ay up) while the second term is the negative phase term (pushing ai down for all i, with weight P(i C).
  - This is an application of a more general technique called importance sampling,

- even exact importance sampling is not efficient because it requires computing weights pi/qi, where pi = P (i C |), which can only be computed if all the scores ai are computed.
- The solution adopted for this application is called biased importance sampling, where the importance weights are normalized to sum to 1.
- When negative word ni is sampled, the associated gradient is weighted by:

$$w_i = \frac{p_{n_i}/q_{n\,i}}{\sum_{j=1}^{N} p_{n_j}/q_{n\,j}}.$$

These weights are used to give the appropriate importance to the m negative samples from q used to form the estimated negative phase contribution to the gradient:

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C)\frac{\partial a_i}{\partial \theta} \approx \frac{1}{m}\sum_{i=1}^{m} w_i \frac{\partial a_{n_i}}{\partial \theta}.$$

A unigram or a bigram distribution works well as the proposal distribution q. It is easy to estimate the parameters of such a distribution from data. After estimating the parameters, it is also possible to sample from such a distribution very efficiently.

- Noise-Contrastive Estimation and Ranking Loss:

  - Sampling-based approaches help reduce the computational cost of training neural language models with large vocabularies.
  - One early method is the ranking loss, which treats each word’s output as a score and tries to rank the correct word higher than negative words by a margin of 1, with the gradient being zero if the margin is satisfied.
  - The ranking loss proposed then is

$$L = \sum_i \max(0, 1 - a_y + a_i).$$

- A limitation of ranking loss is that it does not provide conditional probability estimates, which are needed in applications like speech recognition and text generation.
- A more recent and widely used approach is noise-contrastive estimation, which has been successfully applied to neural language models to efficiently approximate probabilities and reduce computation.

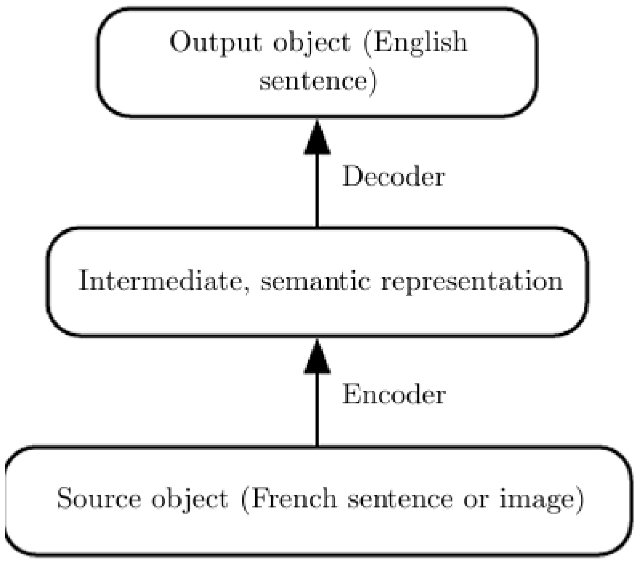
4.Combining Neural Language Models with-grams:

- N-gram models have the advantage of high model capacity with very little computation because they only need to look up a few matching tuples, and using hash tables or trees makes computation nearly independent of capacity.
- Neural networks, in contrast, usually require computation proportional to the number of parameters, although embedding layers and some convolutional networks can add parameters without increasing computation per example.
- One way to increase capacity without excessive computation is to combine neural networks with n-gram models in an ensemble, which can reduce test error if the models make independent mistakes.
- Ensemble predictions can be combined using uniform weighting or validation-set–based weights, and large arrays of models can be used to further improve performance.
- Neural networks can also be paired with maximum entropy models, adding high-dimensional sparse inputs for n-grams directly to the output layer, significantly increasing capacity while keeping additional computation minimal.

5.Neural Machine Translation:

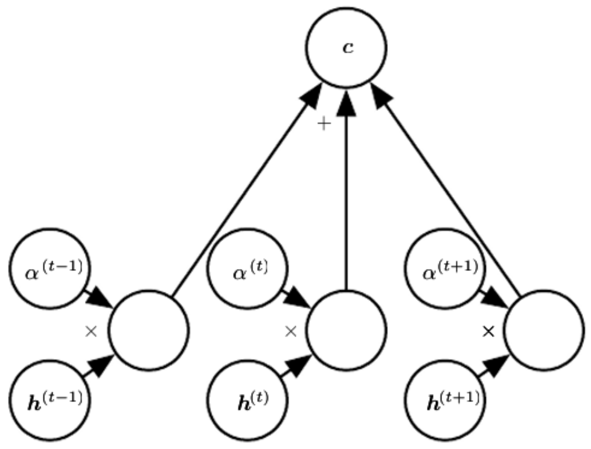
- Machine translation involves reading a sentence in one language and producing an equivalent sentence in another, often requiring a proposal mechanism to suggest candidate translations and a language model to score them for grammatical correctness.
- Early neural network applications in machine translation upgraded the language model component from n-gram models to neural language models, improving prediction of target words given context.

- Conditional language models extend this idea by estimating probabilities of target sequences given source sequences, allowing more accurate translations.


Output object (English sentence)
Decoder
Intermediate, semantic representation
Encoder
Source object (French sentence or image)

- MLP-based models were used to score target phrases given source phrases, but required fixed-length sequences, motivating the use of RNNs that handle variable-length inputs and outputs.
- Encoder-decoder frameworks use one model to read the input sequence and produce a context representation, and another model to generate the target sentence from this context.
- Learning representations where sentences with the same meaning have similar embeddings, regardless of language, improves translation quality.
- Later advances combined RNNs and convolutional networks and scaled models to handle larger vocabularies effectively.
- <u>Using an Attention Mechanism and Aligning Pieces of Data:</u>

    - Capturing all semantic details of long sentences with a fixed-size representation is difficult, even with large RNNs trained sufficiently.
    - A more efficient approach is to read the entire input to get the context, then generate the translation one word at a time, using an attention mechanism to focus on different parts of the input for each output word.



- This attention mechanism, introduced by Bahdanau et al., allows the model to gather the semantic details needed at each step.
- Aligning words in source and target sentences enables relating their embeddings, and learning a translation matrix between word embeddings reduces alignment errors compared to traditional frequency-based methods.
- Earlier work also explored cross-lingual word vectors to support translation tasks.

- We can think of an attention-based system as having three components:

    - A process that “reads” raw data (such as source words in a source sentence), and converts them into distributed representations, with one feature vector associated with each word position.
    - A list of feature vectors storing the output of the reader. This can be understood as a “ memory ” containing a sequence of facts, which can be retrieved later, not necessarily in the same order, without having to visit all of them.
    - A process that “ exploits ” the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).
    -

6.Historical Perspective:

- The concept of distributed representations for symbols was first explored by Rumelhart et al., where neural networks captured relationships between family members using triplets like (Colin, Mother, Victoria).
- This idea was extended to word embeddings by Deerwester et al., initially using SVD, and later learned via neural networks.
- Early neural language models represented words and scaled from small symbol sets in the 1980s to millions of words in modern applications, improving language modeling performance.
- Both character-based and word-based models continue to advance, including modeling individual bytes of Unicode characters.
- Neural language models have been applied to many NLP tasks such as parsing, part-of-speech tagging, semantic role labeling, and chunking, often using multi-task architectures that share embeddings across tasks.
- Two-dimensional visualizations of embeddings became popular with the development of t-SNE, allowing analysis and interpretation of word embeddings.

Other Applications:

- ### Recommender Systems

  - Machine learning in IT is widely used for recommending items to users, including online advertising and product recommendations.
  - These systems predict either the probability of a user action or the expected gain from showing an ad or recommendation.
  - Early recommender systems treated this as a supervised learning problem, predicting clicks, ratings, purchases, or time spent.
  - Collaborative filtering relies on patterns of similarity between users or items to make recommendations.
  - Parametric methods often learn distributed representations (embeddings) for users and items, enabling bilinear prediction using dot products.
  - The bilinear prediction formula combines user embeddings, item embeddings, and bias terms for users and items.

- Embeddings can be visualized in low-dimensional spaces and compared like word embeddings.
- Singular value decomposition (SVD) can factorize the rating matrix into lower-rank matrices to obtain embeddings, avoiding missing data issues.
- The bilinear prediction is thus obtained as follows:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}.$$

- Bilinear prediction and SVD were successfully applied in the Netflix Prize competition, improving recommender systems.
- Restricted Boltzmann Machines (RBMs) were among the first neural network approaches for collaborative filtering and were used in Netflix Prize solutions.
- A limitation of collaborative filtering is the cold-start problem, where new users or items lack historical ratings.
- Content-based recommender systems solve cold-start issues by using extra user or item features, often leveraging deep learning architectures like convolutional networks for rich content such as audio tracks.
- **Exploration Versus Exploitation:**

    - Recommendation problems often resemble **contextual bandits**, a type of reinforcement learning where feedback is only received for the chosen action.
    - Data collected from recommendations is **biased and incomplete**, as we only observe user responses to recommended items.
    - No feedback is available for items not recommended, making it **hard to learn the best possible decisions**.
    - Like reinforcement learning, the learner receives rewards only for the actions it chooses, not for all possible actions.
    - **Contextual band its** consider the context (e.g., user identity) when deciding which action (item) to recommend.
    - The mapping from context to action is called a **policy**, which guides the recommendation system.
    - Reinforcement learning requires balancing **exploration** (trying new actions to gain information) and **exploitation** (choosing the best-known action for reward).

- The **exploration-exploitation tradeoff** is influenced by the time horizon: shorter horizons favor exploitation, longer horizons favor exploration.
- Exploration can be implemented via **random actions** or **model-based approaches** that consider uncertainty in rewards.
- Unlike supervised learning, reinforcement learning **cannot rely on fixed labels** and must infer better actions through interaction.
- Evaluating policies is difficult because the **learner's actions affect the data distribution**, making fixed test sets insufficient.
- Effective contextual bandit strategies aim to **optimize both learning from interactions and maximizing user reward** simultaneously.

Knowledge Representation, Reasoning and Question Answering:

- Deep learning has been highly successful in **language modeling, machine translation, and NLP**.
- Success is largely due to the use of **embeddings for symbols and words**, which capture semantic knowledge.
- Embeddings represent **meaningful information about individual words and concepts**.
- Current research focuses on developing **embeddings for phrases and relationships between words and facts**.
- Search engines already utilize machine learning for such representations, but **further improvements are needed** for more advanced semantic modeling.
- **Knowledge, Relations and Question Answering:**

  - Distributed representations can be trained to capture **relations between two entities**, formalizing facts and interactions between objects.
  - Relations can be represented as **triplets (subject, verb, object)** and attributes as pairs (entity, attribute).
  - Machine learning models require **training data** from unstructured text or structured databases to learn these relations.

- **Knowledge bases** like Freebase, WordNet, or GeneOntology store structured relational information.
- Representations for entities and relations can be learned by treating each triplet as a **training example** and maximizing a training objective.
- Neural language models can be **extended to model entities and relations** by learning embeddings for each relation.
- Relations can be represented in different ways, e.g., **vectors for entities and matrices for relations**, where relations act as operators.
- These models can be used for **link prediction**, predicting missing arcs or facts in a knowledge graph.
- Many knowledge bases are incomplete, so models help **generalize from known facts to new facts**.
- Evaluating models is challenging because missing facts may be **true but unobserved**, so metrics like **precision at 10%** are used.
- Distributed representations and knowledge bases are applied to **word-sense disambiguation**, selecting the correct sense of a word in context.
- Combining knowledge of relations with reasoning and memory mechanisms can help build **question-answering systems**, though full general solutions remain a difficult open problem.