I2DL Study Guide

**Deep Learning**

A subset of machine learning that excels in performance by using multiple neural network layers. Handles high-dimensional data with strong performance via representation learning.

**Machine Learning Formal Definition**

“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.”

And a simple definition Using past examples to do better on a task.

**Example of ML :**

Image Classification, this is Supervised Learning, as the model is trained on labeled data, with X (features), and y (labels), our goal is to learn f(x) that generalizes well to unseen (test) data.

**Types of Learning :**

Supervised, which means the Data has labels, some examples are classification and regression.

Unsupervised, which means no labels, an example is clustering.

**Mathematical Foundation :**

Training Data : for You have n training examples.

Each example has d numeric features.

Label exists in all Real #’s.

Function trying to learn exist in set of possible functions from model.

**Loss :**

**0 – 1 Loss** : Used for classification, just counts wrong predictions.

**L2 Loss (Mean Squared Error)** : Used for regression. L2 Squares the difference between your prediction and the actual value. This ensures positive values and penalizes large mistakes more than small ones.

**Empirical Risk Minimization :**

With machine learning our goal is to find a function f(x) that makes accurate predictions on unseen data. Ideally, we’d minimize how wrong our model is on the real-world data distribution, but we only have access to a finite distribution, our training set, so instead of minimizing the true risk, we minimize the average loss over the training data.

**Machine Learning Pipeline :**

1. Collect Data & Extract Features.

2. Choose a model, Define hypothesis class and loss function.

3. Minimize empirical loss.

**Linear Regression :**

Assuming relationship between inputs and outputs is linear (a straight line), make predictions using a formula multiplying the weights by the features, these weights are the values your model learns during training, they determine how much each feature matters. You find the best weights using MSE to determine loss. You can utilize the normal equation which is purely just solving after plugging numbers in. Gradient Descent is more of a guessing game, where if the dataset is too big, it iteratively guesses, checks how wrong the predictions are, then nudges the weights a little to reduce error. We handle bias by just adding a singe value to each input to shift predictions up or down.

**Maximum Likelihood Estimation :**

For regression, we use L2 Loss (Squared Error), because its smooth and convex and corresponds to MLE. With this we assume there is some model with a specific parameter, we want to pick that parameter so that the probability of seeing our actual data is as high as possible. So, we want to find the parameter that maximizes this likelihood. We use log likelihood for this considering multiplying small probabilities can get very messy.

If you use MLE with a Gaussian Assumption for regression you end up with L2 Loss.

If you use MLE with a Bernoulli assumption for classification, you end up with Log Loss.

**Classification vs Regression :**

Classification : Output is a discrete label.

Regression : Output is a continuous value.

**Decision Boundary :**

Let’s say the model is a weighted sum of inputs, with , each w says how important a feature is. And let’s say that if it fits in class 1, and otherwise it fits in class 0, we can then interpret that hyperplane defined at 0 as a decision boundary. This is a linear classifier, it assumes the data can be separated by a straight line.

**Hinge Loss :**

With wrong predictions we now start to care, if you’re right and confident, zero loss, if you’re wrong or not confident enough, positive loss, this works well with gradient descent.

**Breakdown of Loss Choices :**

0-1 Loss really just tells you if the prediction was right or wrong, but because of this its harder to optimize because its not smooth and non-convex meaning that if we wanted to use gradient descent it wouldn’t work well on it. The solution to this is using L2 loss which works, but still isn’t perfect as it doesn’t really “respect “ the binary nature of the output, so we then have the Sigmoid Function.

**Sigmoid Function :**

This function takes any real number and squashes it between 0 and 1. So it looks like an S Curve. The function is . This now gives us a probabilistic interpretation like “How confident are we that this input is class 1?”. Sigmoid is always symmetric. It is just a smooth version of the step function.

**Logistic Regression :**

Instead of minimizing L2 Loss, logistic regression uses Log Loss (Cross-entropy loss).

**Perceptron :**

We are using the same set up, however, we are training it with a simple rule upon update, if its classified correctly, do nothing, if its wrong, then you set the weight to add the input values, so if something goes wrong you push the weights in the direction that would have made you right. This doesn’t use a loss function. And considering IF the data is linearly separable, the perceptron will converge after a finite number of updates.

**Support Vector Machines (SVM) :**

Perceptron only cares if a point is on the right side of a decision boundary, SVM cares how far a point is from said boundary, that distance is called the margin, a large margin = more confidence and a better generalization. SVM wants to find that separating line that gives the maximum margin. SVM maximizes the magnitude which is related to how far the point is from the boundary. Functional margin alone isn’t reliable unless you normalize the weights. Geometric margin is scale invariant meaning we care about the smallest margin across all points. So the goal with SVM is to find the weight and bias that maximizes the geometric margin. We want to get all points on the correct side of the hyperplane and at least distance 1 away, considering this is a convex optimization problem, its mathematically guaranteed to find the best solution. With this, only the closest points to the decision boundary matter, these are called the support vectors, all the other points don’t affect the final boundary, making SVM’s very robust to outliers. SVM learns the most confident separating line, maximizing the distance between the classes. We can also use Soft Margin SVM where we relax the constraints using slack variables which allow for some violations, we append this to our equation with a regularization parameter that controls the tradeoff with a larger parameter providing less tolerance for error and then vice versa. There also exists Multi-Class SVM, as since SVM is binary we use , One-vs-All (OvA) which is 1 class vs the rest, or One-vs-One which is every pair of classes. Hard margin refers to no misclassifications allowed, soft margin allows for some violations. SVM’s sometimes use Kernel Tricks which allows VM to work with non-linear data by mapping to higher dimensions.

**Overfitting :**

Overfitting happens when your model learns too much detail from the training data, including noise and irrelevant patterns. Like memorizing a practice test rather than actually learning the material. When using linear vs non-linear models, non-linear appears much more appealing allowing for curvier more flexible shapes, but while they can fit the data better they are more prone to overfitting. With something like polynomial regression, as you increase the degree M of the polynomial, a small M will underfit, and not be able to capture the wiggles, a just right M will fit the trend well and a Big M will overfit and follow every bump too closely. So knowing the possibility of this, with a refresher, empirical loss is how well you did on the training data, expected loss is how well it performed on the unseen data, a good model will keep both low, but overfitting means, low training error, and high test error. You can prevent this through, smaller hypothesis class, Occam’s Razor (the simpler model is usually better), More training data, the more data you have the less chance your model is fooled by noise, use prior knowledge, such as if you know that the curve should be linear. Prior vs Experience is a key tradeoff, Prior is what you believe about the world before seeing data, experience is what you learn from data, too much prior might ignore useful patterns, too much experience might overfit to quirks of the dataset. Because training error is biased it doesn’t truly aid in choosing the best model, the solution to this is cross-validation, where you split your data into training and validation sets, where you train on somedata then validate on another part, and you eventually pick the one with the lowest validation error. K-Fold Cross Validation is splitting the data into k chunks, then using each chunk once as validation and others as training then averaging the results which gives you a pretty good estimate of how your model generalizes.

**Regularization :**

Regularization adds a penalty for complexity to your loss function, instead of just minimizing training error you minimize Loss = Training Error + Penalty. There are multiple types of this, the most notable being L2 (Ridge), which the penalty is the square of the weights, which makes the parameters small but usually non-zero, and L1 (Lasso) where the penalty is the absolute value of the weights which encourages that sparsity which means many weights will end up being 0. Think of regularization like weighting your model down so it can’t wiggle too wildly. A bit more general of an explanation of the types of regularization is hard constraint regularization which is where you force the model parameters to stay within a range and Soft Constraint Regularization, where you penalize complexity with an extra term in the loss function. Using lagrangian multipliers, you can show that the hard constraint form is mathematically equivalent to the soft constraint form for some value of gamma, meaning that whether you add a constraint or a penalty, you’re still influencing the optimizer to avoid overly complex models. Regularization = using prior knowledge to guide learning in a Bayesian perspective. The three views are hard constraint which is used if you know the clear limit on the weights, soft constraint, which is where you add penalty to loss, which is most commonly used in L2/L1, and Bayesian which is using priori distributions over parameters, which is useful when modeling uncertainty.

**Multiclass Classification :**

One vs Rest as we explain earlier is really applicable to multi-class classification, consider the idea that you are trying to identify if something is a specific class, like class x or not class x, you can go through each classifier (considering you are working with multiple classes) and check if it is class 1 or not, and after running all of the classifiers, you would pick the class with the highest score or confidence. This is great but can lead to ambiguous predictions if multiple classifiers are confident. One vs One is the process of training a classifier for every pair of classes, so if you have 3 classes, A, B, C, you’ll trin Classifier AB, AC, and BC, and when predicting each classifier votes for a class, and you will pick the class with the most votes. This is often more accurate than OvR but much more computationally expensive.

**Softmax Classifier :**

When you want a single model that handles all the classes at once, you sue something like a Softmax Classifier, which outputs a probability distribution over all classes, and picks the one with the highest probability. Like this input is 70% likely to be A, 20% to be B, and 10% to be C, lets pick A. And when you are working with more than 2 classes now, accuracy alone isn’t enough, you’ll often want to use a confusion matrix, or precision / recall / f1 for each class. Some common mistakes with multiclass, is imbalanced classes, where if some classes are way more frequent, your model might ignore the rare ones, or things like tied scores in OvR, where the model might struggle to choose if multiple classifiers are equally confident, or the training cost of OvO, where lots of classifiers can be overkill for a large k.

**Deep Learning :**

With machine learning we manually extract features, but this brings up the issue that, what if the features aren’t good enough? Representation Learning is the idea that we can learn the features too, not just the model. So with linear models, you just have dot products between the weights and the input, but to handle these higher-degree relationships we need to add some sort of nonlinearity handler, we can do this by using a non-linear activation function like ReLU or sigmoid on a linear transformation to help us find transformations of the data, this allows the network to build its own features. The components of a feedforward neural network are, the input layer (your data as a vector), the hidden layers ( the layers that learn immediate representations using weights and activations), and output layers which give you your predictions. For binary classification you would probably use sigmoid, and for multiclass you would use that softmax. Activation functions are necessary in this process, some of the most common ones are ReLU (which is just the max of 0, z), Sigmoid which squashes input between 0 and 1. Tanh which squashes between -1 and 1. With Sigmoid or Tanh, when inputs get very large or small the output flattens, and gradient then become too small, making learning slow or stuck, that’s why ReLU and its variants are preferred. Like for example with the XOR function, a linaer model cannot separate it properly, but a neural network with a hidden layer and ReLU activations can, as the ReLU introduces that non-lineartiy that allows for complex decision boundaries. A major theorem with deep learning is that , “A feedforward neural network with at least one hidden layer and non-linear activation function can approximate any continuous function – given enough neurons.”. Some key takeaways from this are deep networks learn features and the function, layers that are used are made of linear transformations + non-linear activations.

**Backpropagation :**

Method used to update the weights in a neural network by computing how much each weight contributed to the error. It figures out which weights in the network are responsible for the prediction being wwrong, and then adjusts them accordingly. This works with forward pass as it goes through all the layers, then computes the predictions using current weights, it then computes loss between the predictions and actual labels, the backpropagation then starts here from the loss, working backwards through the network to compute the gradients where is then uses those gradients to update the weights via gradient descent. Some key terms in backprop are Delta, which refers to the error signal for a layer, and how much thi slayer contributed to the loss, Learning Rate which refers to how big the weight updates are. Backprop is what gives neural networks the power to learn from data.

**Other Regularization Strategies :**

Data Augmentation can be used to trick the model into thinking it has more data by slightly modifying the inputs such as rotating, shifting, cropping or adding noise to the images, this also helps with generalization. You can also add noise to inputs which then forces the model to be less sensitive to noise, or to weight whichs simulates training a variety of models, or to outputs to help if they were already potentially noisy or incorrect. Adding noise turns out to be mathematically similar to L2 regularization. Early stopping can also be used, so during training, we monitor validation loss, if it stops improving, we stop training early, it helps as it prevents the model from overfitting the training data. Dropout is also used, which is practically just a random regularization where in each training step you randomly drop some neurons, so only a subset of the model is trained each time, it works as it forces the model to be robust, it can’t rely on any single neuron.

**Convolutional Neural Networks :**

With an example like an image with 1000 x 1000 with RGB means there is 3 million different values, with a normal fully connected neural network, there is too many parameters, CNNs solve this by extracting important local features and reusing parameters and making the model translation invariant. In a CNN, the convolutional layer extracts features, the pooling layer down samples the data to reduce overfitting and computation, and the fully connected layer performs the final classification or regression. Convolution refers to sliding a small filter (kernel) over the input, at each position, we take the dot product o the filter with that region of the image. Stride refers to how far the kernel moves each step and the padding refers to the borders so the output doesn’t shrink. CNNs work so well because of its sparse connectivity, where each neuron only looks at a small region not the whole input, meaning less computation and better generalization, Parameter sharing is also important as the same kernel is applied across different parts of the input meaning there is fewer weights to train. Without padding, convolution shrinks the images, but with padding (same), we can preserve the original size. The goal of pooling layers in CNNs is to reduce the dimensionality while keeping the important info. The types are max pooling, which keeps the max value in a region, and average pooling, which averages the values in that block, pooling helps with translation invariance, which reduces overfitting by reducing the model’s sensitivity to small changes.

**Optimization :**

In normal optimization, you probably know the function you’re trying to minimize, in machine learning you don’t know the true data distribution, so you then focus on minimizing the loss on your training set and hope it generalizes to unseen data. We use empirical risk minimization to minimize the average loss over the training samples. There are problems with ERM In Deep Learning such as neural networks overfitting the training data easily, some of the loss functions not being differentiable. Instead of directly minimizing “did we predict correctly?”, we instead minimize something like negative log likelihood or cross-entropy loss. Early stopping is utilized in deep learning, where you stop early when the validation loss starts increasing to avoid overfitting. We often split the data into mini-batches and then perform gradient descent like that, where each batch is then given an approximate gradient. Optimization is hard in deep learning because of some unique challenges like Ill-conditioning where the loss surface may contain some weird geometry, the solution to this is gradient clipping. Also in neural networks because they are non-convex, there are many local minima, but most local minima are actually good enough, also saddle points are an issue, where gradient is near zero but not a minimum, stochastic gradient descent helps escape these due to randomness. Some of the key optimizations algorithms are SGD (stochastic gradient descent), where each update uses a small batch of data, learning rate is crucial here, too big is unstable and too small is slow, this often reduces learning rate over time. Momentum speeds up SGD by adding a velocity component, this helps escape falat regions and smooths updates. Newtons method, uses both gradients and curvature (second derivatives), it can be very fast, but requires some tricks like regularization. Adam optimizer adapts learning rates for each parameter individually. Optimizing in deep learning is NOT about perfectly minimizing the loss, its about finding a good enough solution that generalizes well, with challenges like overfitting, non-convexity, and exploding gradients they require more special algorithms.

**Factor Analysis :**

A statistical modeling method where we try to represent complex, high-dimensional data using a smaller number of unobserved latent variables. Basically try to explain a bunch of features from a subset of that bunch. Principle Component Analysis is used within this to help us reduce dimensionality while keeping the most important patterns as well as remove the noise from the data. The core idea of PCA is to project high-dimensional data onto a lower-dimensional subspace that captures the most variance. The steps of PCA are to center the data (subtracting the mean), the compute the co-variance matrix to help understand how features vary together, the perform eighen-decomposition on this matrix. The eigenvectors give us the new axes (principal components), the eigenvalues tell us how much variance is captured by each. Then choose the top k eigenvectors which will form our new basic, then project that data onto the new basis. The First Principal Component is the direction that captures the maximum variance in the data, we end up just keeping the first few as each new axis captures less and less variance. PCA is unsupervised, it finds structure based on variance, it can remove noise but also might discard subtle yet useful features, it assumes linear relationships, if non-linear PCA might miss patterns.