03-generalization

HK Turesson

2024-09-17

Contents

Chapter 1

About

This book is based on lecture notes for the Deep Learning course at the Master of Management in AI (MMAI) 5500 at the Schulich School of Business.

How the book was written:

1. Over the years, I built the course content as lecture slides.
2. I recorded myself giving the lecture.
3. The sound recording was transcribed using a speech to text (Whisper, open source?)
4. Regular expression was used to clean up and split the text into sections.
5. Each section was fed into an LLM with a prompt like `Re-write the following text in the style of a concise and not-too-technical introductory textbook in Deeo Learning. Format the output in latex.`
6. The returned texts were checked, corrected, figures were added and combined into lecture notes.
7. These lecture notes were provided to the students together with the slides as course materials.
8. The students were encouraged to edit, comment, ask for clarification, suggest additions etc. Students who contributed in this way were listed as chapter co-authors.
9. These lecture notes became the chapters making up this book.

1.1 Usage

Chapter 2

Applications of Deep Learning in Business

Written by Hjalmar K Turesson

2.1 Goal

The goal with this book is that it should prepare the students for the business world and allow them to do the following:

TODO

Chapter 3

Feedforward Neural Networks

Written by Hjalmar K Turesson, OpenAssistant, ChatGPT and the MMAI class of 2023

TO BE IMPROVED

3.1 Introduction

In this lecture, we will be discussing the fundamental concepts of building neural networks using deep learning techniques. We will begin by exploring the forward pass and its role in processing input data through the layers of a network. Next, we will delve into activation functions, focusing on non-linear activation functions and their significance in shaping the output of each layer.

Afterward, we will examine loss functions and the backward pass, which enables error propagation and gradient calculation throughout the network. Moreover, we will cover the chain rule and weight updates during optimization processes such as Stochastic Gradient Descent (SGD) with appropriate learning rates and momentum. By integrating these essential components, we aim to provide readers with an understanding of how various elements work together to create effective deep-learning models.

3.1.1 Content

3.2 Forward Pass

3.2.1 Introduction

In this section, we will introduce a core type of neural network - the feed-forward neural network. For instance, this process involves taking two inputs, passing them through multiple layers of interconnected artificial neurons, and producing an output. This section aims to provide an overview of the forward pass and help students understand how different neural network components contribute to generating predictions.

We will begin by examining a simple neural network - a single neuron (logistic regression or perceptron) - and how it operates. Building upon this foundation, we will explore multi-layered neural networks and discuss key concepts such as activation functions, weight initialization, backpropagation techniques, including SGD, learning rate manipulation and momentum, and other factors critical to the success of modern deep learning systems.

3.2.2 The neuron

3.2.2.1 The Perceptron and Logistic Regression and Their Relation to Single Neurons

The **Perceptron** is used for supervised learning tasks, particularly binary classification problems.

The Perceptron forms the basis of our exploration of neural networks. Introduced by Frank Rosenblatt in 1957, it is a mathematical model inspired by the biological functioning of animal cells, particularly those found in the retina. Our objective here is to comprehend the behaviour of a single neuron as a first step towards understanding the multi-layer perceptron (MLP).

A perceptron consists of two components: a linear combination function and an activation function. The linear combination function takes the form of a dot product between a set of weight vectors (W), corresponding bias term (b), and input features (x):

$$\hat{y} = \begin{cases} 1 & \text{if } W \times x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\hat{y}$ is the prediction, that is the value returned by the Perceptron.

Note that the Perceptron will always return either 0 or 1 due to the *greater than* ($>$) sign acting as a threshold. Thus, the Perceptron is a *binary classifier*, that

is, it discriminates between two classes. We call this thresholding an *activation function*, and the activation function for the Perceptron is a step function.

More generally, we can write the equation more generally as follows:

$\hat{y} = f(W \times x + b)$

Here, f is the activation function. As mentioned above, in the case of the Perceptron, f is the step function. However, if we replace the step function with the logistic function ($f(z) = 1/(1 + \exp(-z))$) then we get logistic regression where the output ($\hat{y}$) is interpreted as the probability of a class instead of the predicted class itself. That is, logistic regression is a probabilistic classifier.

The logistic function is an example of a sigmoid function. The sigmoid activation function maps the result to a probability ranging from 0 to 1, ensuring that the output falls within this range and can be interpreted as a probablitiy.

3.2.2.2 Summary

Logistic Regression and the Perceptron are similar, with the difference that the former uses the logistic function as activation while the latter uses the step function. Due to the different activation functions the learning rules are also different (not discussed further).

3.3 The cycle of learning

Images of hearts and faces as input X (in reality, the images are made up of more than three pixels), forward pass the input and generate a prediction for the input based on the activation function. In this example, the image is predicted to be a face, while the target image is a heart. This results in an error (aka a cost or a loss). The error is calculated by comparing the network's output to the desired output using a suitable loss function. Starting from the output layer, the gradients are computed layer by layer, propagating the error backwards through the network. For each layer, the gradients are used to update the parameters of that layer.

The training process of a neural network involves iterating through several phases, including forward propagation, error computation, gradient computation, weight updating, and backpropagation. The forward propagation step involves passing the input data through the network's layers and applying weights and activation functions to generate predictions or outputs. Then, the error or loss between the predicted outputs and the true labels is computed. These stages work together to optimize the model parameters and minimize the difference between predicted outputs and true labels. During forward propagation, the input data flows through the network to produce an estimated result, $\hat{y}$, which we subsequently use as ground truth during the backward pass.

The forward pass step-by-step:

1. **Input data**: Start with the input data, which can be a single example or a batch of examples. Each example is typically represented as a vector or matrix.
2. **Input layer**: Feed the input data into the input layer of the neural network. Each input neuron in the input layer corresponds to a feature or dimension of the input data.
3. **Hidden layers**: The input data is then processed through one or more hidden layers of the neural network. Each neuron in a hidden layer receives inputs from the previous layer, performs a linear combination of the inputs using its associated weights, and adds a bias term. A hidden layer is simply a layer that is neither an input or output layer.
4. **Output layer**: The data propagates through the hidden layers until it reaches the output layer. The output layer usually consists of one or more neurons, depending on the task at hand. Each output neuron produces a prediction or estimation based on its inputs and associated weights.
5. **Output**: The final output of the neural network is obtained from the output layer. It represents the network's prediction for the given input data.

$$
\begin{aligned}
h_1 &= f(w_{11}^1 x_1 + w_{21}^1 x_2 + w_{31}^1 x_3) \\
h_2 &= f(w_{12}^1 x_1 + w_{22}^1 x_2 + w_{32}^1 x_3) \\
\hat{y} &= g(w_{1y}^2 h_1 + w_{2y}^2 h_2)
\end{aligned}
$$

During the forward pass, the weights and biases of the neural network are fixed and not updated. The purpose of the forward pass is to compute the output of the network using the current set of parameters. The computed output can then be compared to the ground truth labels or targets to calculate the loss or error, which is used in subsequent stages like backpropagation to update the network's parameters and improve its performance.

3.4 Activation functions

3.4.1 Introduction

This section discusses the challenges of fitting non-linear functions and the benefits of using multi-layer neural networks to approximate any continuous function.

The activation function is applied to the weighted sum to introduce non-linearity and determine the output of the neuron. It maps the weighted sum to the desired output range. Commonly used activation functions in neural networks include the logistic function, hyperbolic tangent (tanh) function, the rectified linear unit

(`ReLU`), and the softmax function. The step function is rarely used in neural networks.

The choice of activation function for output neurons is crucial as it determines the behaviour and interpretation of the network's output.

3.4.2 The problem: fitting a non-linearity

The problem of fitting a non-linearity refers to the challenge of modelling complex, non-linear relationships between inputs and outputs. This problem is evident when attempting to draw the decision boundary of the logical operation exclusive or (XOR), a classic example demonstrating the limitations of single layer neural networks. The XOR problem provides a simple illustration of the need for non-linear decision boundaries, and thus, multi-layer Perceptrons/Neural networks. From Fig 1 it is apparent that it is not possible to draw a linear decision boundary separating the open from the closed circles. While XOR is somewhat artificial and a very simple learning problem, most real-world tasks also require learning non-linear decision boundaries. Thus, for neural networks to have real-world utility, they need to be able to fit non-linearly separable data.

3.4.2.1 The limitation of single-layer neural networks

The activation functions used in neural networks introduce non-linearity. There are three commonly used activation functions for hidden neurons. `ReLU` is a simple function, but introduces non-linearity.

- **Linear Separability**: Single-layer neural networks can only learn linearly separable patterns. They are unable to handle datasets that are not linearly separable, meaning they cannot accurately classify data that requires non-linear decision boundaries. See Figure: Contradictory conditions.
- **Limited Expressiveness**: Single-layer neural networks are limited in their ability to represent complex relationships between inputs and outputs. They can only learn linear transformations of the input data. (To make it clear) The output of a single-layer perceptron can only be binary (i.e. 1 or 0). It restricts the variety of classification.
- **Lack of Feature Hierarchy**: Single-layer neural networks treat all input features equally and independently. In real-world problems, many datasets have high-dimensional inputs where features can have varying degrees of importance and dependencies. Single-layer networks cannot capture such hierarchical relationships.
- **Sensitivity to Input Noise** (e.g singular samples, outliers): Single-layer neural networks are sensitive to input noise. Even small variations in the

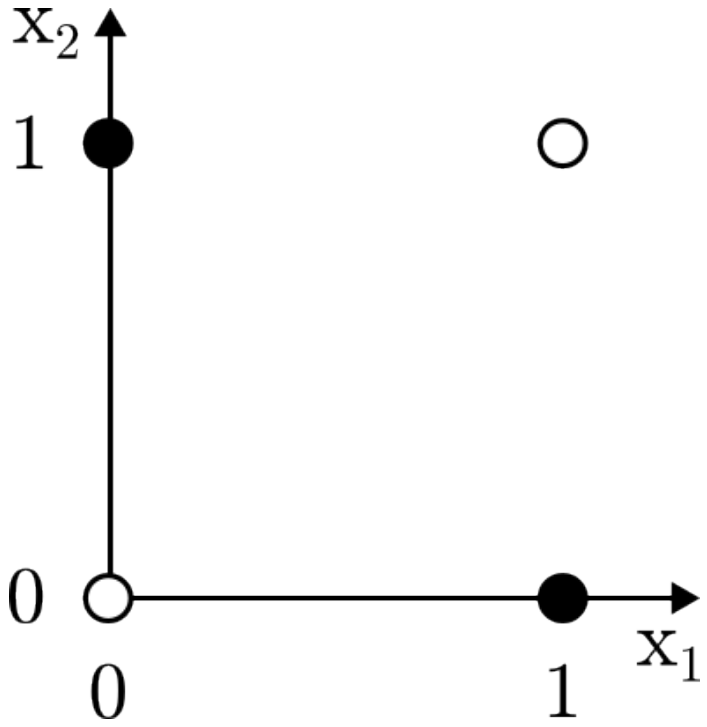

Figure 3.1: Figure 1. The graphical illustration of the exclusive or problem. By H K Turesson.

input can lead to significant changes in the output. This sensitivity makes them less robust in handling noisy or incomplete data. The lack of hidden layers for feature extraction and noise filtering amplifies this limitation.

To solve this issue, more advanced machine learning algorithms were developed, capable of performing non-linear classification tasks. These include Multilayer Perceptrons (MLPs), which use multiple layers of interconnected nodes, allowing them to capture complex relationships and perform non-linear mappings. MLPs also incorporate activation functions that introduce non-linearity within the network, enabling it to solve problems like the XOR problem. Logical OR (||) refers to combining two boolean variables using the 'or' operator, while the XOR problem specifically tests a binary classifier's ability to correctly identify samples belonging to classes defined by the XOR relationship between their features.

The primary motivation for examining activation functions and multi-layer neural networks was to address the limitations of linear models in solving non-linear classification problems. Through a simple example, it became evident that a single perceptron was unable to solve such problems. However, by introducing an additional layer to the network, it became capable of solving the problem effectively. This example demonstrated the potential of multi-layer networks in solving non-linear classification tasks.

The historical significance of this observation dates back to the work of McCulloch and Pitts in 1943, where they showed that multi-layer networks could achieve tasks like XOR classification. In particular, McCulloch and Pitts were the first to determine a mathematical model of a neural network. They published a paper defining these methods in detail. This highlighted that with appropriate weights and network structure, complex problems could be addressed using multiple layers.

While the Perceptron learning rule was a groundbreaking development, allowing networks to learn, it is not applicable to multi-layer neural networks. This presents a challenge as the perceptron learning rule cannot optimize the weights in such networks.

In summary, the investigation of activation functions and multi-layer neural networks stemmed from the need to address non-linear classification problems. While the perceptron introduced learning, it was limited to single-layer networks. The perceptron, as a single-layer network, has limitations in its ability to handle complex patterns and relationships.

The exploration of multi-layer networks and their associated challenges laid the foundation for further advancements in neural network learning algorithms.

The exploration of non-linearities extends beyond classification problems and encompasses various other scenarios, including regression tasks. For instance, when attempting to fit a sine wave within a specific range, a linear network can only provide a suboptimal fit, as indicated by the orange line. The question

arises whether adding more linear neurons or additional layers of linear neurons can improve the fitting of the sine wave. Surprisingly, the answer is no.

Despite increasing the number of linear neurons or layers, a linear network remains incapable of accurately capturing the complexities of a sinusoid. A multilayer neural network with multiple linear activations will still fail to capture these complexities. This limitation highlights the necessity for non-linear activation functions in neural networks, as they enable the network to model and approximate non-linear relationships effectively.

In summary, non-linearities pose challenges not only in classification problems but also in regression tasks. The inability of linear networks to fit sinusoidal patterns emphasizes the significance of activation functions that introduce non-linear transformations, ultimately enabling neural networks to accurately represent and approximate complex functions.

Deep neural networks , have been developed as a solution to overcome the limitations of single-layer networks. By incorporating multiple hidden layers, deep neural networks can effectively learn complex patterns, capture non-linear relationships, and model hierarchical features. This makes them more capable of solving real-world problems with higher complexity and provides greater flexibility in representing and processing data.

3.4.3 Why are non-linear activation functions necessary?

We illustrate a two-layer neural network with linear activation functions. By rearranging the equations governing the computation flow through the layers, we can collapse them into a single layer, showing the limitations of such models without non-linear activation functions. While increasing the number of linear hidden layers wouldn't help capture complex patterns present in non-linear functions, using non-linear activation functions opens up new possibilities to better model data in higher-level representations.

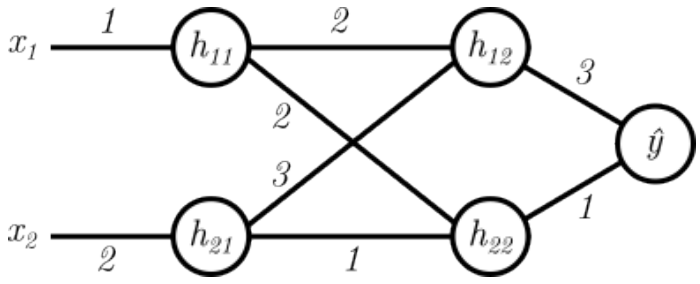

Figure 3.2: Figure 2. A network with three weight layers and linear activations. By H K Turesson.

The Fig 2 above demonstrates that multi-layer networks without non-linear activation functions can be reduced to a single layer. Meaning that it can not

handle non-linear tasks. Non-linear activation functions are necessary for a network to handle non-linear decision boundaries.

$$
\begin{aligned}
h_{11} &= 1x_1 \\
h_{21} &= 2x_2 \\
h_{12} &= 2h_{11} + 3h_{21} \\
h_{22} &= 1h_{21} + 2h_{11} \\
\hat{y} &= 3h_{12} + 1h_{22} \\
\hat{y} &= x_1(1 \times 2) + x_2(2 \times 3 \times 3) + x_2(2 \times 1) + x_1(1 \times 2 \times 1) \\
\hat{y} &= x_1 w_a + x_1 w_b + x_2 w_c + x_2 w_d \\
\hat{y} &= x_1(w_a + w_b) + x_2(w_c + w_d)
\end{aligned}
$$

Neural networks require non-linear activation functions in their hidden layers to enable the approximation of more complex, non-linear functions. Popular choices include the hyperbolic tangent (tanh) and Rectified Linear Unit (ReLU). These functions allow for differentiability and therefore make it possible to learn via backpropagation.

Non-linear activation functions in hidden layers expand the expressive power of deep neural networks, allowing them to approximate more complicated functions and learn more complex features from the data.

In summary, non-linear activation functions are needed to fit non-linear decision boundaries and solve more complex problems.

3.4.4 An example of fitting a non-linear function

In this concrete example, we aim to fit a sine wave using a neural network. The network consists of two layers, each with eight neurons. However, for simplicity, the network is not fully connected.

To illustrate the fitting process, we initially activate only the first units in the two hidden layers, ignoring the others. We gradually add more units step by step to improve the fit of the sine wave.

The weights and biases are assigned specific values. The first weight from the input to the first unit in the first layer is set to 6, resulting in a steep slope. The bias is set to 0. A ReLU activation function is used, represented by a cyan line, to capture the non-linearity of the network.

Next, a weight with a negative slope of 1 is added, illustrated by a negative slope ReLU. This unit has a bias of 0.7. Finally, the last weights have a negative slope of 0.1. These equations describe the activations of each unit, which are calculated by taking the maximum value between the weighted inputs and 0, and then applying the subsequent weights and biases.

The output, denoted as y or $\hat{y}$, is obtained by multiplying the activation of the last unit by the final weight. When plotted, this fitting process approximates

the sine wave with a slope close to 6 initially, followed by a flat region. However, it may not be a perfect fit.

The figure above depicts the process described. Consider the first neuron of the first hidden layer, this neuron has a bias of 0, and it takes in the input (x) multiplied by a weight of 6. Its output is its input put through a ReLU function, being equal to $\max(x \times 6 + 0, 0)$. The output of the first neuron then becomes the input for the second neuron in the next hidden layer. The second neuron's output is the output of the previous neuron, times a weight of -1, plus a bias of 0.7 and then put through a ReLU. This would be equal to $\max(\max(x \times 6 + 0, 0)(-1) + 0.7, 0)$. Finally, the output neuron returns the output of the previous neuron times the weight of -, of course if the output neuron had an activation function associated with it, it would be passed through the activation function. Therefore the final output is $\max(\max(x \times 6 + 0, 0)(-1) + 0.7, 0)(-1)$.

In practice, we wouldn't manually set the weights and biases like we did in this example. Instead, we would start with random weights and biases, then iteratively adjust them using a process called training. During training, we use a dataset of inputs and expected outputs, and an optimization algorithm like gradient descent to minimize the difference between the network's predictions and the true values. This process of training adjusts the weights and biases in the direction that reduces the error.

In summary, this example demonstrates the process of fitting a sine wave using a neural network with specific weights, biases, and activation functions. The gradual addition of units allows for an improved approximation of the desired function, although the fit may not be perfect. The quality of the fit depends on factors such as the architecture of the network, the amount and quality of training data, and the optimization process.

To enhance the fit of the sine wave, we can add the next two units to the network. These units have different settings, including a steeper slope and a bias that shifts the function upwards. By incorporating these units, we achieve a better approximation of the sine wave.

The process of improving the fit can be continued by adding more units and gradually fitting the sinusoid piece by piece. Although the example provided is simplified and hardcoded, it aims to illustrate how adjusting a network's width (number of units) and depth (number of layers) can enable the fitting of a non-linear function.

By strategically modifying the neural network's weights, biases, and activation functions, we can iteratively improve its ability to capture complex patterns and accurately represent non-linear relationships. This highlights the flexibility and power of neural networks in approximating and fitting non-linear functions.

3.4.5 Neural networks are universal function approximators

In the late 1980s and early 1990s, mathematicians Kurt Hornik, Maxwell Stinchcombe, and Halbert White, among others, established that neural networks are universal function approximators. Specifically, they demonstrated that a feed-forward neural network with a single hidden layer – though more layers can also achieve this – can approximate any continuous function within compact subsets of the real numbers. Compact subsets refer to closed (including all their limit points) and bounded (having a finite length, area, volume, etc.) sets on the real numbers. Compact subsets refer to closed and bounded sets within these spaces, not the entire set of real numbers.

This theoretical result implies that a simple neural network (with non-linear activation functions for the hidden layer(s)), equipped with suitable parameters (known as weights and biases), has the capability to represent a wide range of interesting functions, including those that model complex phenomena. It provides proof that, given the right parameters, we can fit and approximate nearly any desired function.

However, finding these "right parameters" is the crux of the challenge in neural network training. The next part of our discussion will delve into the crucial question of how we can effectively learn and determine these appropriate parameters. By understanding the principles and techniques of parameter optimization and learning algorithms, such as gradient descent and backpropagation, and learning algorithms like SGD, Adam, or RMSProp. we can unleash the full potential of neural networks and leverage their universal approximation capabilities.

3.4.6 Activation functions for output neurons

For the output layer in a neural network, the choice of activation function depends on the type of task being performed in a neural network, the selection of an activation function for the output layer is highly task-specific and directly influences the output type:

- **Regression** tasks require a linear activation function to produce numerical outputs (e.g., housing prices). It involves predicting a continuous value; this, this function does not limit the range of output, thus allowing the network to predict numerical outputs of any real-valued number. This requires either a linear or not activation function as outputs need to be allowed to be numbers other than 0 or 1.
- **Binary classification** tasks use a sigmoid or logistic function to provide probabilities for classifying objects as belonging to either Class A or B (e.g., distinguishing cats from dogs). It involves differentiating between two classes. This function squashes the output to a probability between 0

and 1, providing a probabilistic measure for classifying objects as belonging to either Class A or Class B.

- **Multi-class classification** tasks utilize a softmax function to estimate a probability distribution across multiple classes (e.g., identifying different fruits like apples or bananas).The objective is to classify an input into one of several classes. This function generates a probability distribution over K possible output classes, ensuring that the sum of the probabilities equals 1. The predicted class corresponds to the neuron with the highest probability. Therefore the maximum possible value returned by a softmax would be 1, assuming all other probabilities were 0.

Note that output activation functions do not need to be non linear, in fact they must be linear for regression tasks.

3.5 The loss function

In the learning process of a neural network, we follow a cycle that involves the forward pass, generating predictions based on input data. In supervised learning, each prediction is compared to its corresponding target, representing what the prediction should ideally be. This comparison allows us to compute an error, indicating the disparity between the prediction and the target.

The crucial step in reducing the error and improving the network's performance is known as backpropagation. Backpropagation involves using the computed error to determine how the network weights should be adjusted. By propagating the error back through the network, we can calculate the gradient of the loss function with respect to the weights. This provides us with an indication of how to adjust these weights to minimize the error. The gradient guides us in the direction of reducing the error, leading to the update of the weights in a way that minimizes the loss. This is typically done using an optimization algorithm, such as SGD or one of its variants. These algorithms use the computed gradients to adjust the weights and biases, gradually improving the network's performance over successive iterations of training.

However, the speed and quality of learning don't just depend on the calculated gradients. They are also influenced by the learning rate, a hyperparameter that determines how much the weights are adjusted during training. A properly set learning rate allows the model to learn effectively, converging to a good solution. If it's set too high, the model might overshoot the optimal point, while if it's too low, the model could get stuck or learn very slowly. Therefore, tuning the learning rate is a crucial part of training neural networks.

The choice of an appropriate loss function is critical as it determines how the error is quantified and impacts the learning process. Different types of problems, such as regression or classification, require different loss functions tailored to the

specific task at hand. By optimizing the loss function through backpropagation, we aim to iteratively improve the network's performance and enhance its ability to make accurate predictions.

3.5.1 Common loss functions

The error, also referred to as loss or cost, plays a central role in the training process of neural networks. It helps the neural network to adjust its weight when the target values are incorrectly predicted. There are three common cases for defining the loss function, depending on the specific task at hand.

In regression problems, where the goal is to predict continuous values, the mean squared error (MSE) is a commonly used metric for evaluating the prediction error. The MSE is computed as the average of the squared differences between the target values and the corresponding predictions. In this case, both the targets and predictions are real numbers such as price, duration, and quantity. Note: MSE is sensitive to outliers since squaring the errors can magnify their impact on the overall score.

For binary classification tasks, where the objective is to classify instances into two classes, the binary cross entropy loss is commonly used. The prediction, in this case, is a real number between 0 and 1, representing the probability of belonging to the positive class. The target, on the other hand, takes binary values of 0 or 1, where 1 means the instance belongs to the positive class, and 0 means it belongs to the negative class.

In multi-class classification scenarios, where the goal is to classify instances into more than two classes, a generalization of binary cross entropy is employed. The cross-entropy loss considers the probability distribution of the predicted classes, and the target values are represented as categorical variables. The calculating method will be one class versus other classes and run the result multiple times. Then, the probability of each class will be displayed.

These different loss functions capture the nature of the specific problem and guide the learning process by quantifying the discrepancy between predictions and targets. By minimizing the chosen loss function through techniques like backpropagation, neural networks can be trained to make more accurate predictions and improve their overall performance.

3.5.1.1 Cross entropy

Cross entropy, also known as categorical cross entropy, is a common loss function used in multi-class classification tasks. In this case, the targets (y_{true}) are represented as one-hot encoded categories, and the predictions (y_{pred} or $\hat{y}$) are the softmax outputs or probability distributions over the classes. The softmax output is a vector of the same length as the target (one-hot encoded) vector

where each value is between 0 and 1 and their sum is 1. The cross-entropy loss quantifies the dissimilarity between the predicted probability distribution and the one-hot encoded target. It measures how well the predicted probabilities align with the true class label. For example, if we have a classification task between three classes (say cats, dogs and penguins), softmax will return a vector with the probabilities of an example belonging to a particular class. Thus, if we input an example: $[1, 0, 0]$ indicating the input is a cat (1 for cat, 0 for dog and 0 for penguin) and after going through our network, we get the softmax output as $[0.7, 0.1, 0.2]$. This would imply that our network predicts that the probability of the input belonging to the first class (cats class) is 0.7, that for the second class (dog class) is 0.1, and that of belonging to the final class (penguin) is 0.2.

Fun fact: The sum of output softmax probability values is equal to 1 (This is because these probabilities are exclusive).

In Python and NumPy, computing cross-entropy is straightforward. By using the negative log-likelihood, the formula for cross-entropy can be written succinctly:

```
import numpy as np
def cross_entropy(y_true, y_pred):
    epsilon = 1e-12  # small value to avoid division by zero
    y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)  # clip values to avoid numerical
    cross_entropy = -np.sum(y_true * np.log(y_pred)) / N
    return cross_entropy
```

The `y_true` and `y_pred` are arrays representing the target and predicted distributions, respectively. The `np.clip` function is used to ensure that the predicted probabilities are within a valid range (avoiding values close to zero or one). The negative log likelihood is then computed by taking the element-wise multiplication of the target and predicted arrays, followed by summation. In this case the `y_true` are one hot encoded values.

By minimizing the cross entropy loss during training, neural networks can learn to produce probability distributions that closely match the true class labels, leading to improved accuracy in multi-class classification tasks. The lower the loss in cross-entropy means higher accuracy of the model, i.e, if the loss is as close to the value 0 the model's accuracy increases.

3.6 The backward pass

The backward pass, also known as backpropagation, is the step in the learning cycle where we compute the gradients to determine how to update the weights of the neural network in order to minimize the error. The backpropagation is a step-by-step algorithm to minimize the difference between the predicted output and the target.

The backpropagation algorithm plays a vital role in iteratively updating the neural network weights to minimize the error by computing gradients and adjusting the network's parameters.

During the backward pass, the gradients are computed by propagating the error backwards through the network. This is achieved by applying the chain rule of calculus to compute the derivatives of the loss function with respect to the weights of each layer. By iteratively adjusting weights based on gradients calculated during backward pass, the algorithm allows the network to learn from training data and improve its predictions over time.

See chain rule formula.

See Gradient descent vs backpropagation.

The gradients indicate the direction and magnitude of the weight updates that will reduce the error. By iteratively adjusting the weights in the opposite direction of the gradients, the neural network learns to improve its predictions. The partial derivatives for every parameter make up the gradient, the gradient which tells us how the loss changes when we change the parameter.

Backpropagation efficiently calculates these gradients by using the concept of computational graphs and storing intermediate values during the forward pass. The gradients are then propagated backwards through the graph to update the weights. Gradient is the directional vector which tells us in which direction to go and how to increase the loss. To decrease said loss, as mentioned above, we should adjust the weights in the direction OPPOSITE to the gradients.

This process allows the neural network to iteratively learn and refine its internal representations to better align with the target outputs. The gradient points towards higher values, which is why there is a change in the direction of the arrows. In order to compute the gradient, we need softmax or take derivatives of the activation function.

By combining the forward pass, which generates predictions, with the backward pass, which computes the gradients, the neural network can continually update its weights to improve its performance on the given task.

3.6.1 The gradient

The gradient is a vectors that contains the partial derivatives of the loss function with respect to the parameters of the neural network, such as weights and biases.

$$\nabla f(x, W) = \begin{bmatrix} \frac{\partial E}{\partial y} \\ \cdots \\ \frac{\partial E}{\partial w_{21}^{1}} \end{bmatrix}$$

The above equation shows the gradient as a vector of partial derivatives.

They provide information about how the loss changes when the parameters are adjusted. In simple terms, the minimum loss equals the best prediction. It's used to calculate whether to increase or decrease the weights and also to calculate the loss produced due to this change of weights. Hence we use this to minimize the loss as much as possible to build a good model for our tasks. This can be represented in the below equation:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta^t}$$

Above θ^t represents all parameters (weights and biases) at time step t, X the input and the learning rate is represented by alpha (α).

For only the weights:

$$W^{t+1} = W^t - \alpha \frac{\partial E(X, \theta^t)}{\partial W^t}$$

For only the biases:

$$b^{t+1} = b^t - \alpha \frac{\partial E(X, \theta^t)}{\partial b^t}$$

During the learning process, the goal is to minimize the loss function, which represents the discrepancy between the predicted output of the neural network and the actual target output. By computing the gradients, we can determine the direction and magnitude of the parameter updates that will lead to a decrease in the loss.

Based on the step size of the learning rate, the parameter reaches the point of convergence where the loss function is at its lowest and most optimal

Each element of the gradient vector represents the rate of change of the loss with respect to a specific parameter. These elements are the partial derivatives of each and every parameter as well as biases. Positive values indicate that increasing the parameter will result in a higher loss, while negative values indicate that increasing the parameter will lead to a lower loss.

Loss landscape tells you the direction to go to increase/decrease the loss. The length of the line also tells us the size of the movement in a particular direction. The gradient points in the direction of the steepest ascent, meaning it points toward higher loss. However, during the learning process, we want to minimize the loss, so we take the negative gradient (also known as the negative gradient direction) to move in the direction of the steepest descent, where the loss decreases. This is highly dependent on selecting an appropriate learning rate (denoted by alpha (α). A value that's too high may overshoot and fail to converge, while a value too low might take too long to converge (as well as be computationally expensive).

The figure above shows the relation between gradient and loss. The darker the gray, the more loss is incurred.

Since neural networks typically have multiple parameters, the gradient is a vector because it includes one partial derivative for each parameter. By updating the parameters in the opposite direction of the gradient, we can iteratively minimize the loss and improve the performance of the neural network.

Click here for an in-depth explanation.

3.6.2 Derivatives of the activation functions

The derivatives of activation functions are important in computing the gradients during the backward pass of neural network training. Here are the derivatives of some commonly used activation functions:

1. **Relu** (Rectified Linear Unit, $\max(0, x)$)
    - Derivative: It is 1 when the input (x) is greater than zero and 0 otherwise.
    - Relu is the most commonly used activation function.
2. **Sigmoid**
    - Derivative: The derivative of the sigmoid function (σ) is a function of the sigmoid itself.
    - Derivative formula: $\sigma(x)(1 - \sigma(x))$
3. **Softmax**
    - Derivative: The derivative of the softmax function depends on the specific implementation and the overall loss function being used.
    - It is typically computed as part of the overall gradient calculation for the loss function.
    - The result after the Softmax will be between 0 and 1.

Knowing the derivatives of activation functions allows us to compute the gradients of the loss function with respect to the parameters, which in turn enables us to update the parameters during the learning process.

3.6.3 How to calculate the gradient

To calculate the gradient during the backward pass, we use the chain rule from calculus. In the context of neural networks, we can think of the forward pass as a chain of functions. Let's consider a simplified neural network with one hidden unit ($h(\cdots)$) and two weights (w_1, w_2) and gradient function $g(\cdots) = w_2 f_1(w_1 x)$.

The output of the hidden unit can be represented as a composite function, denoted as capital $F(\cdots) = h(g(\cdots))$, which is equal to the output of function h, applied to the output of function g. Function g takes the input x and multiplies it by weight w_1, while function h takes the output of g and multiplies it by weight w_2.

The derivative of this composite function, capital F, can be calculated using the chain rule. The derivative, denoted as capital F' or the gradient of F, is equal to the derivative of function h with respect to the output of g, multiplied by the derivative of function g with respect to its input.

By applying the chain rule, we can break down the computation of gradients for each layer of the neural network, propagating the gradients backward through the layers. This allows us to efficiently compute the gradients of the loss function with respect to the parameters and update the weights accordingly during the training process.

In summary, the gradient represents the direction and magnitude of the change required to minimize the loss function. It is useful to determine how the parameters should be updated to improve the network's performance.

3.6.4 Backpropagation

During the backpropagation process, we go through seven steps to compute the gradients and update the weights of the neural network.

Let's summarize these steps:

1. Start at the output layer and compute the gradient of the loss with respect to the output node. This represents the partial derivative of the error with respect to the output node, $\partial E/\partial \delta$.

2. Use the gradient from step 1 to calculate the error with respect to the input of the output node, which we call $\partial E/\partial zy$. Here $\partial E/dz = \partial E/\partial y \times \partial y/\partial z$.

3. Compute the gradients of the weights in the second layer, denoted as $\partial E/\partial w_{21}y$ and $\partial E/\partial w_{22}y$, using $\partial E/\partial zy$ from step 2. Here $\partial E/\partial w = \partial E/\partial z \times \partial z/\partial w$.

4. Calculate the error with respect to the output of the hidden layer nodes, $\partial E/\partial h_1$ and $\partial E/\partial h_2$, by using the partial derivatives from step 3 and the weights connecting the hidden layer to the output layer. Here: $\partial E/\partial h = \partial E/\partial w \times \partial w/\partial h$

5. Compute the gradients of the inputs to the hidden layer nodes, $\partial E/\partial zh_1$ and $\partial E/\partial zh_2$, using $\partial E/\partial h_1$ and $\partial E/\partial h_2$ from step 4. Here $dE/dz_1 = \partial E/\partial h \times \partial h/\partial z_1$.

6. Calculate the partial derivatives of the error with respect to the weights in the first layer, denoted as $\partial E/\partial w_{11}^1$, $\partial E/\partial w_{12}^1$, $\cdots$ and $\partial E/\partial w_{32}^1$, using $\partial E/\partial z h_1$ and $\partial E/\partial z h_2$ from step 5.

7. Update the weights by subtracting the learning rate α multiplied by their respective gradients. The learning rate determines the step size we take in the direction opposite to the gradient, aiming to reduce the loss.

By going through these steps, we can compute the gradients for all the parameters (weights) in the network and update them accordingly to minimize the loss. Note that this description assumes no biases for simplicity, but biases can be incorporated in a similar manner.

3.6.5 Local minima

Gradient descent, as a form of local search, is not guaranteed to find the global minimum of a function. It can get stuck in a local minimum instead. The concept is illustrated in a figure showing a curve with two local minima, one of which is also the global minimum. While one of the local minima coincides with the global minimum, gradient descent may converge to the other local minimum, unable to escape its vicinity and find the optimal solution. Consequently, in the presence of multiple local minima, gradient descent may not always reach the global minimum, and alternative optimization techniques may be necessary to overcome this issue.

3.6.6 Parameter updates

The final step in the process is performing parameter updates (a.k.a optimization) by adjusting the weights and biases according to their corresponding gradients derived from the training data to improve model performance. This completes the overall learning cycle where the model learns to predict target variables by optimizing its parameters during the learning phase using supervised learning techniques.

3.6.6.1 Stochastic gradient descent

The standard and most common optimization method is called SGD. It involves taking the old weights and subtracting a scaled version of the gradient. SGD historically referred to updating the parameters per sample, but nowadays, it typically refers to mini-batch gradient descent, where updates are performed on mini-batches of data.

Mini-batch gradient descent is the most commonly used method today, where the parameters are updated after each mini-batch, but 'mini-batch' is just called

a batch today (i.e. the method is called batch gradient descent, or often SGD). The mini-batch size can vary, typically ranging from 32 to 1024 samples or more, depending on the size of the dataset. The average gradient over the mini-batch is computed, and the parameters are updated accordingly. The mini-batch size increases depending on what the computational resources permit.

Batch gradient descent, on the other hand, traditionally meant a single update of the parameters on the entire dataset by calculating the average gradient across the dataset. However, this approach is less commonly used today.

In summary, stochastic gradient descent now commonly refers to mini-batch gradient descent, where updates are performed on batches of data rather than individual samples.

3.6.7 Improvements to SGD: Learning rate decay

The learning rate decay is an improvement to SGD that involves gradually reducing the learning rate as the training progresses. This technique is inspired by the concept of simulated annealing, where larger steps or higher learning rates are initially used to explore the optimization space and quickly converge towards promising regions, and then smaller steps are taken to fine-tune the parameters and avoid overshooting.

By starting with a higher learning rate and gradually decreasing it over time, the learning rate decay allows for efficient optimization. It helps to prevent the model from getting stuck in suboptimal solutions or overshooting the minima. As training progresses and the parameters get closer to the optimal values, smaller steps ensure more precise updates and better convergence.

Learning rate decay can be implemented in various ways, such as using a fixed schedule where the learning rate is reduced at specific intervals or using adaptive methods that dynamically adjust the learning rate based on the progress of the training process or the loss value.

In summary, learning rate decay is a technique used in stochastic gradient descent to gradually reduce the learning rate during training, allowing for more effective optimization and improved convergence toward optimal solutions.

3.6.8 Improvements to SGD: Momentum

Momentum is an improvement to SGD that introduces a moving average of gradients over multiple updates. This moving average is referred to as momentum, mimics the concept of inertia or momentum in physics. It helps to smooth out the fluctuations in gradient updates and enables the optimizer to continue in the direction of the accumulated momentum.

When the gradients consistently point in the same direction over consecutive updates, the momentum increases, allowing the updates to gain speed and overcome small obstacles. This behaviour is similar to a ball rolling down a hill, gradually picking up speed and rolling over minor bumps. It is essentially weighting the gradient descent towards the direction in which the gradient is decreasing.

By incorporating momentum, the optimizer is less likely to get stuck in unfavourable local minima and can navigate through regions of flat gradients more efficiently. It helps accelerate the training process and enables the optimizer to find better local minima.

Momentum can be seen as a complementary approach to learning rate decay. While learning rate decay adjusts the step size to avoid overshooting the minima, momentum helps the optimizer overcome obstacles and push through regions of shallow gradients.

In summary, momentum is a technique used in SGD to introduce a moving average of gradients, which acts as inertia or momentum. It allows the optimizer to continue in the direction of the accumulated momentum, helping to avoid getting stuck in local minima and accelerating the training process. Combined with learning rate decay, momentum can lead to faster convergence and improved optimization.

Chapter 4

Minimizing the Generalization Error

Written by Hjalmar K Turesson, ChatGPT, and the MMAI class of 2023

4.1 Introduction

This chapter covers the following topics: capacity, overfitting and underfitting. It also discusses various regularization techniques to deal with the issues that arise from the aforementioned. Examples of regularization techniques are data augmentation, dropout, parameter tying and sharing, and early stopping. Finally, the issues of overfitting and the importance of evaluating models on both the test set and the evaluation set are discussed.

For further reading, the book Deep Learning by Aaron Courville, Ian Goodfellow, and Yoshua Bengio, specifically Chapter 7 on regularization for deep learning, is recommended. This book is authored by renowned deep-learning researchers and can be accessed freely online. For additional information, various podcasts can help increase the understanding of what to expect for the future state of AI. Examples of such podcasts include the Robot Brains podcast by Peter Abbeel and the Lex Friedman podcast. These podcasts include conversations about and with industry leaders in the field of AI and are recommended.

4.1.1 Content

- Capacity, overfitting and underfitting
- Regularization
- Data augmentation

- Droput
- Parameter tying and weight sharing
- Early stopping

4.2 Capacity, overfitting and underfitting

4.2.1 The goal of machine learning

The goal of machine learning (ML) is to ensure that *the model performs well on new, unseen data* by minimizing the generalization error or the out-of-sample error. The generalization error refers to the expected error on new inputs drawn from a distribution representative of real-world scenarios. The generalization error is *estimated* by the test error, that is, the error on the *test data*.

4.2.2 The goal in two parts

Minimizing the generalization error involves two aspects: *minimizing the training error* (variance) and *minimizing the difference between the training and test error* (bias). Only focusing on minimizing the training error can lead to overfitting. This will cause the model fit noise in the training data and fail to generalize well in unseen data (e.g. test data). However, minimizing the difference between training and test error without also minimizing the training error leads to underfitting. Over and underfitting are the two central challenges in ML, and whether a model over- or under-fits depends on it's *capacity*.

4.2.3 Model capacity

Model capacity refers to a model's ability to fit a wide variety of functions. A low-capacity model struggles to fit the training data effectively, while a high-capacity model can even learn noise or irrelevant patterns specific to the training data that are absent in unseeen data. It is important that the model capacity fits the complexity of the task.

Factors such as the model's architecture and number of parameters influence its capacity. Over the past few years, deep learning models, especially transformer-based architectures, have shown notable advancements in NLP tasks by utilizing larger model capacities together with increasing amounts of data.

4.2.4 Examples of model capacity

In the context of classification, the decision boundary can be used as an example to understand model capacity (in regression the fit will similarly illustrate a model's capacity).

TODO: *On the left in the figure below, we have a linear decision boundary, which represents a low-capacity model that is underfitting the data. In the middle, we have a curved decision boundary, which represents a moderate capacity model capable of fitting the data well. On the right, we have a very complex decision boundary, indicating a high-capacity model that may overfit the data. In the low capacity and high capacity, the key tasks are to find the architecture or learners that would match the complexity of the model.*

Similarly, in regression examples, the same principle applies—models with too low or too high capacity may fail to accurately capture the underlying patterns, while an appropriate capacity model performs well.

4.2.5 Model capacity illustrated

If we plot model capacity on the x-axis [from low to high capacity, as we move right] and error on the y-axis [from low to high error, as we move up] (see Fig 1), we observe a similar pattern to model training. As capacity increases, training error decreases until it becomes stable. This is as expected because a larger capacity model indicates a more complex model, which will fit the training set better. We can also see that until the orange line, the generalization error decreases. This is due to the model learning.

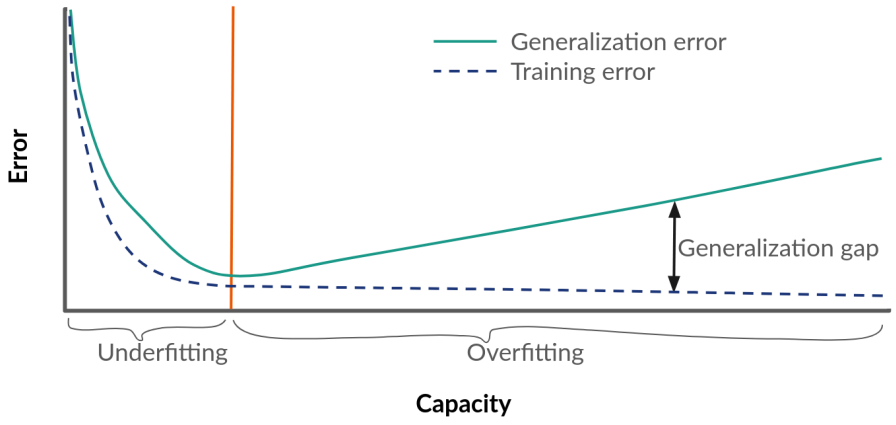

Figure 4.1: Figure 1. By HKT.

However, as we keep increasing the capacity and complexity of our model, we see the generalization error increase as well. This is due to the model becoming so complex that it starts fitting noise. The difference between generalization error and training error represents the generalization gap.

4.2.6 How to minimize the generalization gap

To minimize the generalization gap, several approaches can be employed:

1. **Reduce the number of features**: This is commonly used in linear regression, where the number of input variables is reduced to avoid overfitting. $L1$ regularization could be used for feature selection and thus reducing the number of features. $L1$ regularization encourages sparsity by pushing some weights to exactly zero. Features connected to zero-weights are assumed to be unimportant and thus removed. $L2$ regularization differs in that it penalizes large weights more than $L1$, leading to a more distributed reduction of the weights across all features.

2. **Decrease model capacity**: In neural networks, reducing the number of parameters in the model can help prevent overfitting. A model with too little capacity cannot learn the problem, whereas a model with too much capacity can learn it too well and overfit the training dataset. Both cases result in a model that does not generalize well. The Chinchilla scaling law is an example of finding the optimal balance between model capacity and dataset size for large language models.

3. **Reduce the magnitude of the parameters** aka **weight decay**: $L1$ or $L2$ regularization techniques can be applied to reduce the magnitude of parameters in the model. This is particularly useful when multiple features contribute slightly to the prediction, and removing any single feature is not feasible. In addition to $L1$ and $L2$, Elastic net regularization is a linear combination of the $L1$ and $L2$ penalties.

4. **Early stopping**: Iterative training can be stopped when the validation error starts increasing, preventing the model from overfitting to the training data.

5. **Increase data quantity**: Having more data is an effective, if not the best, way to improve generalization.

6. **Data augmentation**: A technique used to artificially increase the amount of training data available for a model. This works for classification and object recognition but is also effective for speech recognition. Some of the popular image augmentation techniques are flipping, translation, rotation, scaling, changing brightness, and adding noise.

7. **Dropout**: Dropout is a technique where randomly selected neurons are temporarily dropped (set to zero) out during training, which helps prevent over-reliance on specific features and encourages the model to learn more robust representations.

8. **Weight sharing**: Weight sharing involves sharing parameters between different parts of the model, such as in convolutional neural networks

(CNNs) and recurrent neural networks (RNNs). This method indeed helps reduce the complexity of the model and improve generalization.

4.3 Regularization

Regularization, as defined by Corville, Goodfellow, and Bengio in their book Deep Learning, encompasses any modification made to a learning algorithm with the aim of reducing generalization error but not its training error. It is a broad concept that does not only include weight decay, as seen in $L1$ and $L2$ regularization. Examples of regularization techniques include constraints and penalties designed to encode specific kinds of prior knowledge, designed to give preference for simpler models, necessary to make under-determined problems determined, and combining multiple hypotheses as done with ensemble models.

4.3.1 Weight decay

Weight decay comes in two forms, $L1$ regularization and $L2$ regularization, also known as *ridge regression* or *Tikhonov regularization*. Both involve adding a penalty term to the loss function that depends on the magnitude of the parameters.

During the training process, neural networks learn to minimize a loss function that quantifies the discrepancy between the predicted output and the true output. Weight decay is an additional term added to the loss function that penalizes large weights in the network. This regularization technique helps to control the complexity of the model, reduces overfitting, and improves its ability to generalize well to unseen data.

Typically, only the weights are regularized, while the biases are left unaffected. The reason for this is that weights require more data to accurately fit, whereas biases require less data. Fitting a weight depends on observing both the upstream and downstream neurons, whereas a bias in only connected to one neuron, and thus requires comparatively less data to fit accurately. This makes biases contribute less to overfitting.

4.3.1.1 $L1$

$L1$ regularization, commonly used in LASSO regression, uses the sum of the absolute values of the weights as the penalty term.

$$L1 = \lambda \sum |\theta_i|$$

It results in more sparse solutions, where many weights are set to zero. The derivative or slope of the $L1$ regularization penalty is always either -1 or 1,

indicating a constant drive to minimize it as long as the weight is non-zero. As a result, $L1$ regularization tends to drive many weights to exactly zero.

$$\frac{\partial L1}{\partial \theta_i} = \lambda \begin{cases} 1 & \text{if } \theta_i \geq 0 \\ -1 & \text{if } \theta_i < 0 \end{cases}$$

4.3.1.2 $L2$

$L2$ regularization, used in Ridge regression, uses the sum of the squares of the weights as the penalty term.

$$L2 = \lambda \sum \theta_i^2$$

This penalty function forms a parabolic curve, and as the coefficient approaches zero, the penalty term's gradient becomes less steep, resulting in coefficients that are small but often not strictly zero.

$$\frac{\partial L2}{\partial \theta_i} = \lambda 2 \theta_i$$

This means that as the weight gets closer to zero, the drive or penalty to further decrease it diminishes. Thus, $L2$ regularization, many weights tend to be close to zero but not exactly zero, as the penalty gradually decreases but does not completely eliminate them.

4.3.1.3 When to use which?

$L1$ regularization leads to sparser solutions with more weights set to zero, while $L2$ penalizes large parameters more and tends to produce weights close to zero but not exactly zero.

$L1$ regularization is useful when you want to perform feature selection, as it tends to push the coefficients of less important features to zero. This can result in a more interpretable and efficient model.

$L2$ regularization is useful when the goal is to to prevent overfitting and improve the generalization of the model.

It's noteworthy that when dealing with correlated features, L1 regularization tends to select one feature and discard the others, while L2 regularization will keep all correlated features but distribute the coefficient weights among them.

4.4 Data augmentation

Data augmentation is a technique used to increase the amount of training data by creating modified or artificial examples from the existing data. It is employed when obtaining more real-world data is not feasible.

Data augmentation is commonly used in classification tasks, particularly for object recognition, but it has also shown effectiveness in speech recognition. The reason data augmentation works better with classification tasks (than say, regression) is because it is easier to learn/obtain properties of non-continuous classes. For example, in a cats vs dogs classifier, it is easier to augment data of a particular class by playing around with the properties of each class (rotating images, re-scaling images, etc). For regression tasks, when augmenting data, it is very difficult to know for certain which property contributes to what extent to the target.

By augmenting the data, the model can learn from a more diverse set of examples, improving its generalization performance. Data augmentation helps in mitigating the problem of overfitting by introducing variability and reducing model dependency on specific features. Moreover, in the context of deep learning, a certain form of data augmentation can be implemented as part of the model itself, acting as a form of regularization, thus making models more robust to slight variations in the input data (see Dropout).

4.4.1 Image data augmentation

Image data augmentation is used in computer vision tasks, such as image classification or object detection, to increase the diversity and size of image data sets. It involves applying various transformations and manipulations to the images to create new augmented examples (see Fig. 2 for examples).

The taxonomy of image data augmentation methods includes basic image manipulations such as kernel filters, colour space transformations, random noise, geometric transformations, and mixing of images. There are also deep learning approaches such as adversarial training, neural style transfer, and further augmentation methods like meta-learning and auto augmentation (see Fig 3.).

The selection of augmentation methods depends on the specific task, dataset, and desired variations to enhance the model's generalization capabilities. Although not discussed in detail here, image data augmentation is an area of ongoing research and development in machine learning.

4.4.1.1 Image data augmentation – geometric transformations

Geometric transformations are a type of image data augmentation that involves applying transformations such as rotation, flipping, and zooming to an origi-

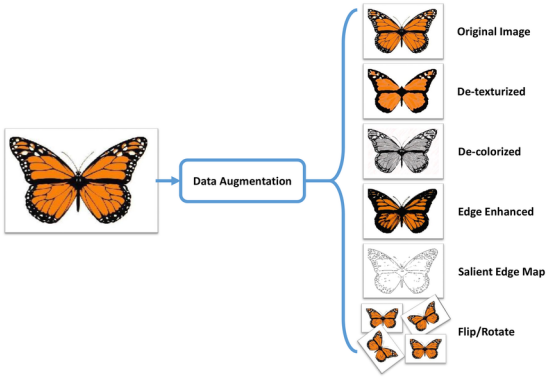

Figure 4.2: Figure 2. By Ahmad, Muhammad and Baik from Data augmentation-assisted deep learning of hand-drawn partially colored sketches for visual search.

nal image. In the provided example (see slide 25 TODO), the top row shows examples of rotations where the image is rotated by different degrees to the left or right. The second row demonstrates flipping the image upside down and the third row showcases zooming together with cropping the image at different locations. By using these geometric transformations, it is possible to generate multiple new augmented images from a single original image, as seen in the example where eight additional images are created from one original image. However, geometric transformations are unsuitable in certain tasks, like optical character recognition (OCR). If we, for example, rotate or reflect characters like p and q, it will mess inadvertently change the class label.

4.4.1.2 Image data augmentation – colour augmentations

Colour augmentation is another type of image data augmentation that involves modifying the colours and contrast of an image. Examples of this are shown in Fig 2 and Fig 4. In the provided examples, there are various color augmentation methods demonstrated.

In Fig 4 contrast modification, histogram equalization (adjusting the distribution of pixel values in the image), white balance adjustment (the color temperature of the image is modified) and sharpening are demonstrated. These color augmentation techniques provide diverse image variations that can be used for data augmentation.

4.4.1.3 Image data augmentation – kernel randomization

Kernel filtering is a type of image data augmentation where pixels are randomized within a kernel. A kernel is a small square of pixels. The size of the kernel

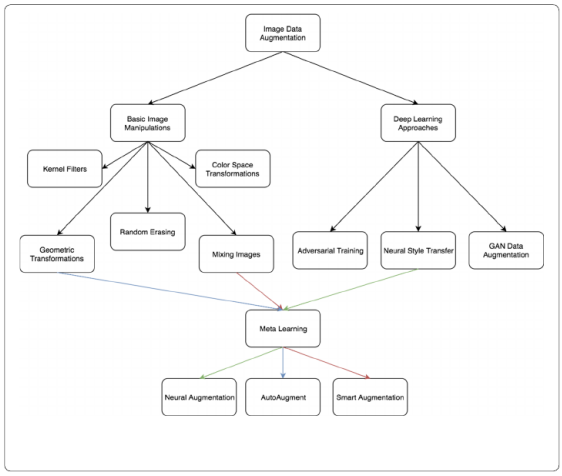

Figure 4.3: Figure 3. By Shorten and Khoshgoftaar from A survey on Image Data Augmentation for Deep Learning.

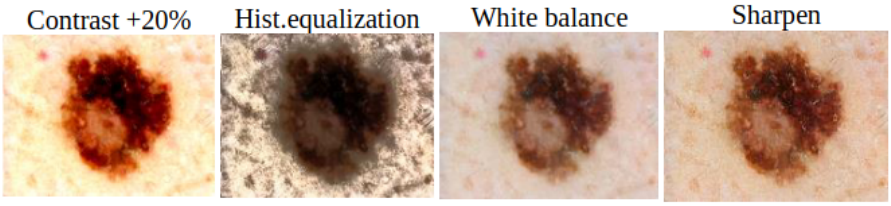

Figure 4.4: Figure 4. Color augmentation for melanoma (a type of skin cancer) classification. By Mikołajczyk and Grochowski from Data augmentation for improving deep learning in image classification problem.

can vary, such as 2×2, 4×4, 6×6, or 8×8. By randomizing the pixels within the kernel, edges and contrast are locally smeared while the objects or features in the image remain in the same location, allowing the same label to be applied.

Increasing the kernel size leads to augmented images with higher levels of noise/smearing, resulting in more pronounced modifications and potential visual distortion.

4.4.1.4 Image data augmentation – noise

Image data augmentation can be done by introducing noise to an image, resulting in diverse augmented images that retain the original label. In some cases, adding small amounts of random noise to the data can be beneficial, as certain tasks like object classification and regression should be robust to noise.

Noise can also be introduced to hidden neurons in neural networks. This is called done dropout [see Dropout], where randomly selected hidden neurons are deactivated, which can, at a high level, seen as applying a multiplicative noise mask to the input data.

4.5 Dropout

Dropout is a technique introduced by Geoffrey Hinton's lab in 2012 in a paper titled Improving neural networks by preventing co-adaptation of feature detectors. It can be seen as a computationally efficient approximation to bagging in neural networks and also as a form of data augmentation (see Image data augmentation – noise).

Dropout works by randomly disabling a certain fraction of the neurons during the forward pass. This means that certain neurons are temporarily set to zero. The dropout rate or dropout probability is a hyperparameter that controls the fraction of masked neurons.

Dropout can be thought of as approximating the effect of training an ensemble of sub-networks by removing hidden neurons from the network. It has been shown to work better than other regularization techniques, such as weight decay. Dropout can also be combined with other forms of regularization and is compatible with various architectures and training methods.

4.5.1 Steps to use Dropout when training

1. *Select a Dropout Rate*: The dropout rate is the probability of the activation of a neuron being set to zero. Common dropout rates are between 0.2 and 0.5.
2. *Select layers to apply Dropout to*: Dropout is applied per layer in a neural network. You can apply dropout to input layers as well as hidden layers.
3. *Train with Dropout*: During training, activations are randomly set to zero within each batch according to the dropout rate. This means that each update to a layer during training is performed with a different "view" of the downstream layer.
4. *Scale activations at during prediction*: After training, when making predictions, dropout is no longer used and instead all neurons are active. However, to compensate for the greater number of active neurons, the outputs of the dropout layers have to be scaled. This is typically done by multiplying the layer output by the retained rate (i.e., $1/p_{dropout}$).

4.5.2 Dropout and prediction

During prediction or inference, we want to use the average output of all sub-networks rather than a single sub-network or mask. This approach is known as ensemble averaging or model averaging. This is similar to the concept of averaging models in bagging, more specifically, the average of multiple predictions will be more accurate than one prediction. There are different ways to achieve this. One approach is to directly average the output of multiple masks, but this would require multiple forward passes, which can be computationally expensive.

A better option is to use weight scaling. During the forward pass without dropout (no masks), the weights are scaled by the inverse of the dropout rates. This scaling is necessary to prevent the output from being too large. For example, if the dropout rate is $p_{dropout} = 0.5$ (50% of neurons are set to zero), the weights are scaled by $1/0.5 = 2$ to compensate for the fact that, on average, only half of the neurons were active during training and thus their weights had to be bigger to produce the same output. This approach allows us to approximate the ensemble prediction without the need for multiple forward passes.

4.6 Parameter tying and sharing

In situations where multiple networks or sub-networks learn mappings from similar input and output distributions, parameter tying and sharing can be employed as regularization techniques. It encourages the parameters of the networks to be similar to each other, promoting shared learning and facilitating generalization. These approaches help in reducing the complexity of the models and improving generalization by encouraging the networks to learn common representations or constraints.

Parameter tying involves penalizing the difference between two sets of parameters, such as set θ^A and set θ^B, using a regularization term like $L2$ penalty.

$$\Omega(\theta^A, \theta^B) = (\theta^A - \theta^B)^2$$

The goal is to encourage the parameters of the two networks to be similar to each other, though not necessarily identical, allowing some degree of flexibility in the learned representations.. The squared difference between the parameter sets is used as a regularization term during training.

Parameter tying occurs in *multi-task learning*, when multiple related tasks need to be learned simultaneously. Parameter tying can leverage shared information, which facilitates knowledge transfer between tasks.

Parameter sharing, on the other hand, requires sets of parameters to be equal. This means that the same weights are used on different parts of the input.

$$\theta^A = \theta^B$$

Parameter sharing is a regularization technique commonly employed in neural networks. This technique is commonly used in recurrent neural networks (RNNs), convolutional neural networks (CNNs), and Autoencoder architectures. Parameter sharing reduces memory usage and is computationally efficient. In CNNs, parameter sharing is often applied through the use of shared filters or kernels. These filters are convolved across different spatial locations of the input, allowing the network to capture local patterns and spatial dependencies. By sharing the same weights across multiple locations, the model can exploit the spatial invariance of the data, reducing the number of unique parameters and enhancing computational efficiency. Additionally, this sharing of weights enables CNNs to generalize well to new examples with similar spatial structures.

Both parameter tying and parameter sharing are regularization methods that help constrain the model and improve generalization performance. They encourage similarity or equality among the model's parameters, enabling better transfer of knowledge and preventing overfitting. These techniques are particularly beneficial when dealing with similar input and output distributions, allowing multiple networks to leverage shared information for more effective learning.

4.7 Early stopping

Early stopping is another regularization technique popular deep learning. It involves monitoring both the training loss and the validation loss during training.

Typically, the training loss continues to decrease as the model learns, while the validation loss initially decreases but eventually starts to increase. The goal is to stop training when the validation loss starts increasing, indicating that the model's generalization has reached it's peak.

To implement early stopping, the parameters of the model are saved whenever the validation loss improves. If the validation loss does not improve for a specified number of epochs or batches (e.g., five epochs), training is stopped, and the model is reverted to the parameters that achieved the best validation loss.

Early stopping is popular in deep learning because it leads to better models and saves training time by avoiding unnecessary epochs that would only result in worse validation loss. Since most of the large models, for example, Large language models, require a lot of money to be sent for each epoch, this technique can save considerable resources. To utilize all data (also the validation data), the network can be re-trained on the entire data set (no validation set) for the number of epochs determined by early stopping.

A drawback is that it can be sensitive to the validation set. A validation set that isn't a good representative of the test data can lead to poor results.