**Week 9: Deep Neural Network**

Importance of non-linearity

* Without the nonlinearity, the whole network just computes a linearly weighted sum and these are equivalent
* Nonlinearity is crucial , but a number of options exist, all similar to sigmoids
* ReLU (Rectified Linear Unit) is the most popular

**Loss function**

Loss(Ypred, Ytrain)

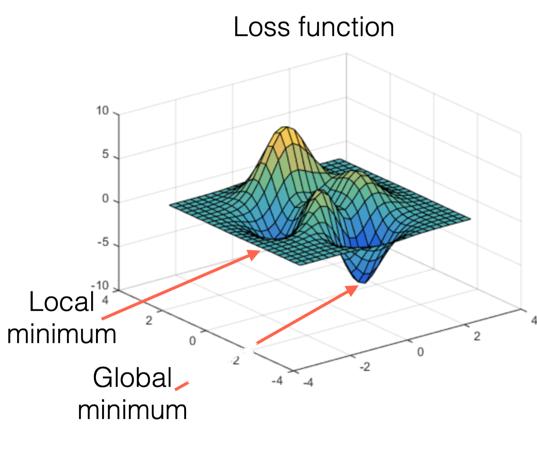
Ypred = g(xtrain)

* Minimise loss by changing parameters (weights and biases)
* Sum over batches/epochs

**Optimisation**

* Used to find minimum loss value
* Typically use methods based on gradient descent: backpropagation
* Important to have right: Initialisation, learning rate

A picture containing text, plot, line, diagram

Description automatically generated

Baches and epochs

* Optimiser calculates gradients and updates once per batch
  + Batch size = number of samples
    - Impacts memory requirements
    - Impacts execution speed
* Repeat baches for the whole training set
  + One epochs = all training samples
* Repeat epochs until convergence

Loss and activation functions: regression

Loss function options: mean\_squared\_error and mean\_absolute\_error

Activation function for final node:

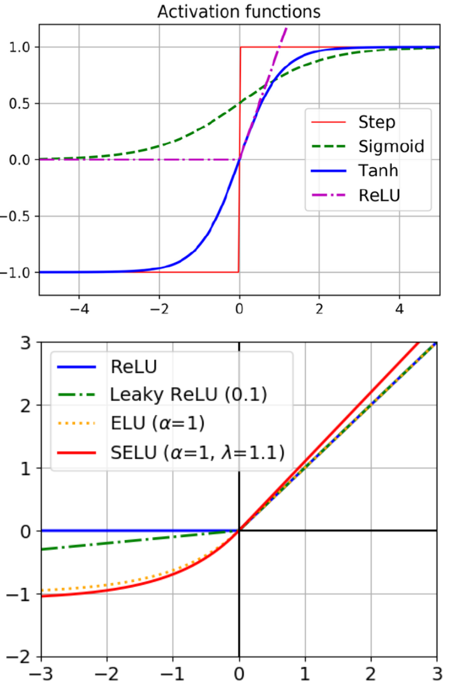
* None (unconstrained)
* ReLU (positive range)
* Sigmoid (limited range)

Loss and activation functions: classification

Loss function options:

* binary\_crossentropy
* categorical\_crossentropy
* sparse\_categorical\_crossentropy
  + Each prefers extreme values (near 0 or 1)

Summary

* Network architecture: fully connected layers, one to next
* Neurons: importance of nonlinear activation functions (ReLU)
* Parameters: weights and biases, number of parameters
* Loss functions, epochs, batches, optimisers
* Regression:
  + loss = mean squared error ; mean absolute error
  + activation = None, ReLU, sigmoid
* Classification:
  + loss = cross entropy variants
  + activation = softmax (with one-hot representation)

**Week 10: Training Deep Neural Networks**

**Non-saturating Activation Function**

Many options:

* sigmoid saturate for high/low values
* ReLU: for negative values, becomes 0 and stays 0 (with 0 gradient)
  + Encourages spareness
* Leaky ReLU: non-saturating, non-dying
* Exponential linear unit (ELU): average closer to 0, converges faster, slower to compute
* Scaled exponential linear unit (SELU): self-normalises dense sequential NNs

**Batch normalisation**

* To prevent growing or shrinking gradients through layers
* Add normalisation layer before or after each hidden layer that learns optimal mean and scale for each input of a layer

**During training:**

1. Standarise to mean 0 and standard deviation 1 across current training batch
2. Scale with adjustable parameter Y
3. Shift with adjustable parameter B
4. Create moving average across batches

**Initialisation strategies**

* Initialise connection weights randomly with mean=0
* Aim to “strategically” have weights that keep signal variance the same
* Keras default: Glorat (with normal distribution)

A picture containing text, screenshot, font, number

Description automatically generated

A picture containing text, screenshot, number, font

Description automatically generated**Optimisers: Baseline (Stochastic) GD**

Gradient Decent

Idea: Follow the gradient downhill

* Need to pick a step size
* Not always the most efficient
* Variations exist to improve on this
* Often uses physics for inspiration

Options are:

* Adam (Adaptive Momentum Est.)
  + Combine momentum and scaling
* Nadam
  + Adam with Nesterov calculation

**Early stopping: Shorter runtimes**

A screenshot of a computer code

Description automatically generated with low confidence

**Learning rate**

A picture containing text, screenshot, font, line

Description automatically generated

**Assignment 3 Notes on Functions and Concepts**

**Perceptron:**

The perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see Figure 10-4) called a threshold logic unit (TLU), or sometimes a linear threshold unit (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. Just like in logistic regression, the model parameters are the input weights w and the bias term b.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a fully connected layer, or a dense layer. The inputs constitute the input layer. And since the layer of TLUs produces the final outputs, it is called the output layer.

**Batch Normalisation:**

A technique that consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.

**Alpha Dropout:**

a variant of dropout that preserves the mean and standard deviation of its inputs. It was introduced in the same paper as SELU, as regular dropout would break self-normalization.

**Optimizers**

**SGD (Stochastic Gradient Descent):**

picks a random instance in the training set at every step and computes the gradients based only on that single instance. working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration. due to its stochastic nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. Once the algorithm stops, the final parameter values will be good, but not optimal.

**Fast Optimizers**

**Adam:**

stands for adaptive moment estimation, combines the ideas of momentum optimization and RMSProp. Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter η.

**Nadam:**

Is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam.

**Activation**

**Swish:**

The Swish activation function is a non-linear activation function. It is designed to be a smooth alternative to other activation functions such as ReLU and sigmoid. The Swish activation function has a similar shape to the ReLU function but with a smoother curve. It allows the activation to take on both positive and negative values, making it more flexible in capturing complex patterns in data. Additionally, the smoothness of the function helps in gradient-based optimization during training.

The main advantage of the Swish activation function is that it tends to preserve more useful information in the network compared to ReLU, especially for deep neural networks. It has been observed to provide improved performance and better generalization on a variety of tasks, including image classification and natural language processing.

**Relu:**

The ReLU (Rectified Linear Unit) activation function is defined as f(x) = max(0, x), which means that it outputs the input value if it is positive and outputs 0 if it is negative. The ReLU function is simple, computationally efficient, and has been shown to work well in many applications. However, one limitation of ReLU is that it can cause "dead neurons" where a neuron outputs 0 for all inputs, which can limit the performance of the network.

**Selu:**

The scaled exponential linear unit (SELU) is an activation function used in artificial neural networks. It was introduced as a self-normalizing activation function that helps alleviate the vanishing/exploding gradient problem and improve the training of deep neural networks. It has the unique property that it can maintain the mean activation close to 0 and the standard deviation close to 1, even in deep neural networks. This self-normalizing property allows for stable gradients and faster convergence during training.

**Softmax:**

The softmax activation function is commonly used in the output layer of neural networks for multi-class classification problems. It takes a vector of real numbers as input and outputs a probability distribution over multiple classes. The softmax function is often used in the final layer of a neural network to generate class probabilities. The output of the softmax function can be interpreted as the likelihood of the input belonging to each class. The class with the highest probability is typically chosen as the predicted class.

One of the benefits of the softmax function is that it provides a smooth and differentiable representation of class probabilities, making it suitable for optimization algorithms that rely on gradient-based methods. Additionally, the probabilities generated by softmax can be easily interpreted and used for decision-making.

**Kernel Initializer**

**He Normal:**

Is a weight initialization technique commonly used in deep neural networks. is specifically designed for activation functions that have a rectifying nature, such as ReLU (Rectified Linear Unit) and its variants. It aims to address the vanishing/exploding gradient problem and promote stable training in deep networks.

The He Normal initializer initializes the weights of each layer with random values drawn from a Gaussian distribution with a mean of 0 and a standard deviation calculated as sqrt(2/n) where n is the number of inputs of the layer.

By using the He Normal initializer, the weights are initialized in a way that keeps the variances of the forward and backward propagations approximately the same, preventing the gradients from vanishing or exploding during training. This helps improve the stability and convergence of the network.

**Lecun Normal:**

Is a weight initialization technique. It initializes the weights of each layer with random values drawn from a Gaussian distribution with a mean of 0 and a standard deviation calculated as sqrt(1/n) where n represents the number of inputs in a layer. By using the Lecun Normal initializer, the weights are initialized in a way that ensures the activations neither saturate too quickly nor are too small, leading to more stable gradients during training. This can help prevent the vanishing gradient problem commonly associated with deep networks using saturating activation functions.

**Early Stopping:**

a regularization technique used in training neural networks to prevent overfitting and improve generalization performance. It involves monitoring a validation metric during training and stopping the training process early when the performance on the validation set starts to deteriorate. The basic idea behind early stopping is that as training progresses, the model learns both the underlying patterns in the data and the noise or idiosyncrasies present in the training set. Over time, the model may start to overfit the training data, meaning it becomes too specialized and fails to generalize well to unseen data. Early stopping helps prevent this by monitoring the validation metric, which serves as a proxy for the model's generalization ability.

**Model Checkpoints:**

Model checkpointing is a technique used in training neural networks to periodically save snapshots of the model's parameters during the training process. These snapshots, known as checkpoints, capture the state of the model at different points in time and can be used for various purposes, such as resuming training, evaluating the model's performance, or deploying the model for inference.

**MC Dropout:**

a technique that utilizes dropout during both training and inference in neural networks to estimate uncertainty and make more robust predictions. Dropout is a regularization technique commonly used during training, where randomly selected neurons are temporarily "dropped out" or ignored during forward and backward passes. during inference, instead of using a single forward pass through the network, multiple forward passes are performed with dropout enabled. This is done by randomly dropping out neurons according to the dropout probability used during training. The outputs of these forward passes are then averaged to obtain an ensemble prediction.

The main motivation behind Monte Carlo Dropout is to estimate the uncertainty associated with predictions. By performing multiple forward passes with dropout enabled, the network is exposed to different subnetworks or configurations due to dropout, resulting in diverse predictions. The variance or spread of these predictions can be used as an estimate of uncertainty.

**Learning Rate Scheduling:**

a technique used in training deep neural networks to adjust the learning rate over the course of training. The learning rate determines the step size taken during gradient descent optimization, affecting how quickly the model converges or escapes from local minima. The goal of learning rate scheduling is to find an optimal balance between quickly making progress early in training while also allowing for fine-tuning and convergence as training progresses. A fixed learning rate may not be optimal throughout the entire training process, and learning rate scheduling techniques aim to adapt the learning rate dynamically based on certain criteria.

**Exponential Decay Schedule:**

a common learning rate scheduling strategy used in training deep neural networks. It gradually reduces the learning rate over time by applying an exponential decay factor to the initial learning rate.

The general formula for exponential decay is as follows:

*learning\_rate = initial\_learning\_rate \* decay\_rate^(epoch / decay\_steps)*

Exponential decay schedule is a widely used technique as it provides a simple and effective way to reduce the learning rate during training. By gradually decreasing the learning rate, it allows the model to make smaller updates to the weights as it approaches convergence, leading to more precise optimization and potentially better generalization performance.

**Using Pre Trained Models (MobileNetV3Small):**

A family of lightweight convolutional neural network (CNN) architectures designed for efficient and fast inference on mobile and embedded devices. MobileNetV3 is an evolution of the previous MobileNet models, incorporating improvements in performance and accuracy. MobileNetV3-Small is specifically designed for resource-constrained environments where computational resources, memory, and power consumption are limited. It aims to provide a good balance between model size, accuracy, and efficiency.

A picture containing text, screenshot, font

Description automatically generated

**Formulas**

**Number of Batches per Epoch (probably should know):**

*Number of batches per epoch = (Total number of samples in the dataset) / (Batch size)*

**\*Parameters within a 2D convolution layer (should definitely know):**

*((Kernel height \* kernel width \* no. of inputs) + 1) \* number of kernels*

**Calculate the output size in a convolution layer (probably should know)**

*((Input Size – Kernel Size + 2\*Padding) / Stride) + 1.*

\*Stride is equal to 1 if not specified.

**\*Calculation of the amount of parameters (weights) in a neural network (should definitely know):**

*(Amount of inputs in current layer + 1 bias) \* (amount of inputs in next layer)*

\*See diagram below, this formula is repeated and summed until the output layer is reached. This summing of the layers will give you the total amount of parameters in a DNN model. The formula itself is the calculation of the amount of parameters in a given layer.

A diagram of a network

Description automatically generated with medium confidence

**Good to know Formulas (probably not required but no guarantee)**

**Exponential Decay Schedule:**

*learning\_rate = initial\_learning\_rate \* decay\_rate^(epoch / decay\_steps)*

**Output of a Threshold Logic Unit:**

*output = { 0, if input < threshold*

*1, if input >= threshold }*

The activation function used in the TLU is the Heaviside step function, which produces a binary output of 1 if the input is greater than or equal to 0, and 0 otherwise.

**Perceptron Learning Rule:**

*Δw (change in weights) = η (learning rate) \* (true - prediction) \* x (input feature)*

The perceptron learning rule updates the weights based on the difference between the target output and the predicted output. If the prediction is correct, the weights remain unchanged. However, if the prediction is incorrect, the weights are adjusted in the direction that brings the prediction closer to the target.

**Calculation of variance for initialization of weights in a fully connected layer**

Variance = 2 / (n\_in + n\_out)

N\_in is the amount of input connections

N\_out is the amount of output connections

**Example calculation of the output of a weighted neuron with ReLU activation (most likely should know as well).**

To calculate the output of the neuron with weights [0.2, -0.5] and a bias of 0.1 using the ReLU activation function, we first compute the weighted sum of the inputs and the bias:

Weighted sum = (0.2 \* 1) + (-0.5 \* 1) + 0.1 = 0.2 - 0.5 + 0.1 = -0.2

Next, we apply the ReLU activation function to the weighted sum. Since the ReLU activation function sets negative values to zero, the output of this neuron would be:

Output = max(0, -0.2) = 0