Chapter 3 – Optimization

Fabien Petit

Department of Economics, University of Barcelona
Centre for Education Policy and Equalising Opportunities, UCL

Introduction

Networks can approximate almost any function, but they have many parameters and require careful optimization

- This lecture covers a variety of tips and tricks to **optimize large network models** successfully
- Optimization algorithms, activation functions, parameter initialization, and regularization techniques
- Develop an intuitive sense of how these elements impact optimization and interact with each other

Better Learning

Better Learning

Recall the **gradient descent update rule** for a model parameter

$$\beta_t = \beta_{t-1} - \eta \nabla \beta_t \tag{1}$$

where η is the learning rate and $\nabla \beta_t$ the gradient

- Standard methods to compute the optimal step are computationally prohibitive (e.g., Newton-Raphson)
- We can afford large steps at the start, but they should become smaller as we approach the minimum
- Better algorithms implement an adaptive or an individual learning rate or a combination of both[1]

[1] Learning rate schedulers can also be used to adjust the learning rate depending on the epoch and obtain better convergence.

Better Learning

Figure: Local minima

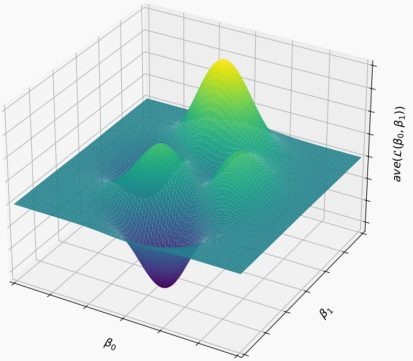
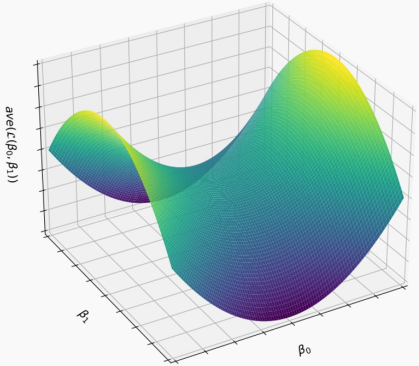


Figure: Saddle point



Local minima are smaller curvatures on the function graph where the optimization can converge. Saddle point appears in two or more dimensions; it is a minimum with respect to one parameter and a maximum with respect to another. In both cases, the gradients are near zero.

Momentum

The **momentum algorithm** (Qian 1999) accumulates the gradient of the past iterations to compute the step

$$\beta_t = \beta_{t-1} - \underline{\gamma\left(\eta\nabla\beta_{t-1}\right)} \times \eta\nabla\beta_t$$

where the tuning parameter γ controls the dependence on the previous accumulated gradients (e.g., $\gamma = 0.9$)

- This causes gradient descent to accelerate when multiple iterations point in the same direction
- The exponential moving average causes the weights of the previous gradients to decay with each iteration

Momentum

$$
\begin{aligned}
\Delta\beta_0 &= 0 \\
\Delta\beta_1 &= \eta\nabla\beta_1 \\
\Delta\beta_2 &= \gamma\Delta\beta_1 + \eta\nabla\beta_2 = \gamma\eta\nabla\beta_1 + \eta\nabla\beta_2 \\
\Delta\beta_3 &= \gamma\underline{\Delta\beta_2} + \eta\nabla\beta_3 = \gamma\left(\underline{\gamma\eta\nabla\beta_1 + \eta\nabla\beta_2}\right) + \eta\nabla\beta_3 \\
&= \gamma^2\eta\nabla\beta_1 + \gamma\eta\nabla\beta_2 + \eta\nabla\beta_3 \\
\Delta\beta_4 &= \gamma\underline{\Delta\beta_3} + \eta\nabla\beta_4 = \gamma\left(\underline{\gamma^2\eta\nabla\beta_1 + \gamma\eta\nabla\beta_2 + \eta\nabla\beta_3}\right) + \eta\nabla\beta_4 \\
&= \gamma^3\eta\nabla\beta_1 + \gamma^2\eta\nabla\beta_2 + \gamma\eta\nabla\beta_3 + \eta\nabla\beta_4 \\
&\vdots \\
\Delta\beta_t &= \gamma^{t-1}\nabla\beta_1 + \gamma^{t-2}\eta\nabla\beta_2 + \gamma^{t-3}\eta\nabla\beta_3 + \cdots + \eta\nabla\beta_t
\end{aligned}
$$

Momentum

Momentum accelerates when gradients for multiple iterations point in the same direction and slows down otherwise

- Fast in areas with a gentle slope (e.g., plateau) and can escape saddle-points and some local minima
- **Faster is not always better**, as Momentum may repeatedly overshoot the minimum before converging
- Momentum does not correct for the direction, as the same learning is applied to every parameter

RMSProp & Adam

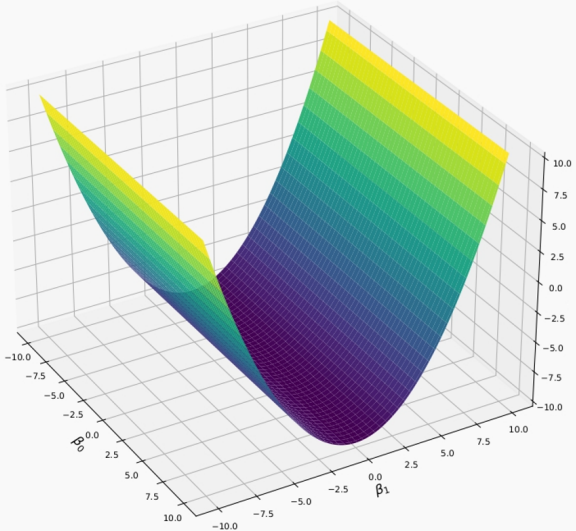


Figure: Corridors

Corridors are a common feature of loss landscapes. To navigate them efficiently, different learning rates should be applied to each parameter (i.e., small for β_0, large for β_1). We can use the fact that gradients are larger in the β_0 direction and smaller in the β_1 direction.

RMSProp & Adam

Root Mean Square Propagation or **RMSProp** (Hinton 2012) adjusts for the direction using individual learning rates

$$\beta_t = \beta_{t-1} - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla \beta_{t-1}$$
$$v_t = \gamma v_{t-1} + (1 - \gamma)(\nabla \beta_{t-1})^2$$

where v_t is a moving average and ε is for numerical stability

- Sets an individual learning rate for each parameter, more robust and oscillates less than Momentum
- Adaptive Momentum Estimation (Kingma and Ba 2014) combines Momentum and RMSProp[2]

[2]The popular AdamW optimization algorithm (Loshchilov and Hutter 2017) also includes a form of L_2 regularisation.

Better Learning

A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam) by Lili Jiang

Batches

Mini-batch gradient descent performs gradient descent steps with a random partition of training observations

- The training data is shuffled and partitioned into a number of mini-batches without replacement
- Each optimization iteration updates the parameters using a mini-batch rather than the entire training sample
- When all mini-batches have been used to update the parameters, an epoch has been processed

Batches

At each step, the computed gradients are different from those of the loss with the entire training sample

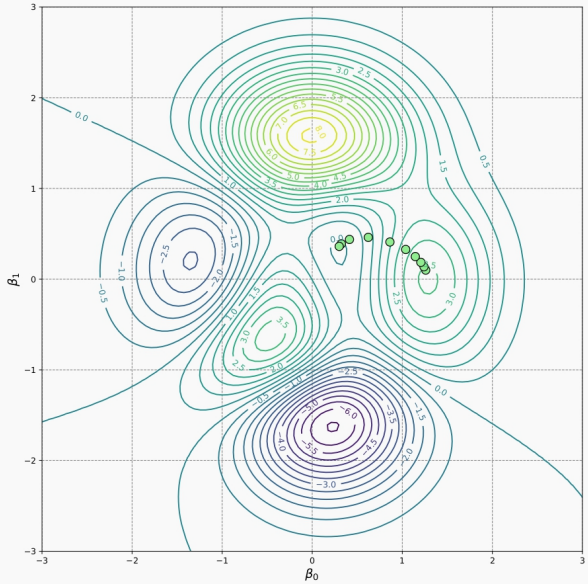
- Noisy optimization path steps are not exactly taken in the direction of steepest descent with regard to the entire sample
- This may prevent optimization from getting stuck in local minima or saddle points for multiple iterations
- Optimization requires more steps, but the updates are computationally cheaper (i.e., fewer observations)

Batches

The mini-batch size is a trade-off between safety with regard to local minima, optimisation stability and speed of convergence

- For simple loss functions or datasets without much redundancy, gradient descent performs well
- For large and redundant datasets, mini-batches are computationally cheaper and provide good estimates
- Standard values for the mini-batch size are 32 or 64 (i.e., powers of 2 for GPU), along with small learning rates

Batches



Better Activation

Better Activation

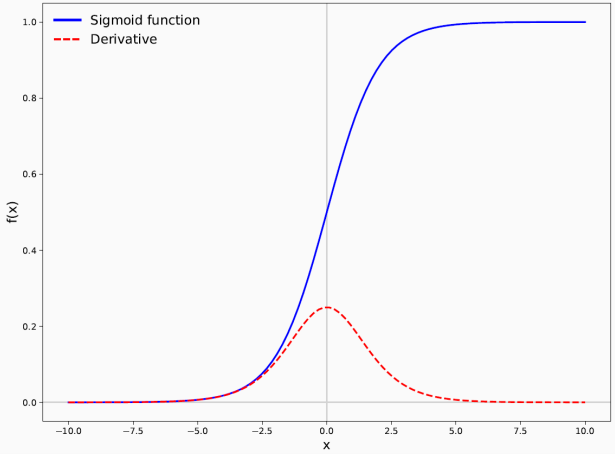
With identity activation, the network would simplify to a linear model, so non-linear activations are essential

$$x_{iu}^{(l)} = \sigma^{(l)}\left(z_{iu}\right) = \sigma^{(l)}\left(x_i^{(l-1)} \cdot \beta_u^{(l)}\right)$$

where σ is the **activation function**

- The activation of the output layer depends on the target distribution (e.g., linear, logistic, Poisson)
- Since hidden units have no target output, the activation is chosen for optimization purposes

Sigmoid



The sigmoid transformation was used in early networks. However, it has a small range, isn't zero-centered, and is relatively expensive to compute because of the exponential

Sigmoid

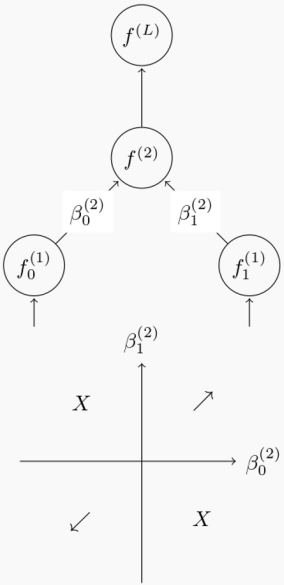
Recall that the backpropagation equations use the derivative of the activation function for each unit

$$\frac{\partial x_{iu}}{\partial z_{iu}} = \sigma'(z_{iu}) = \sigma(z_{iu})(1 - \sigma(z_{iu}))$$

where layer indexes are dropped for simplicity

- When z_{iu} is large, either positive or negative, the sigmoid function's output is close to 0 or 1 (i.e., saturation)
- Given that $z_{iu} = x_i \cdot \beta_u$, this could happen when the elements of x_i or β_u are large (more on this later)
- The partial derivative approaches 0, so the parameters are no longer updated with new observations

Sigmoid



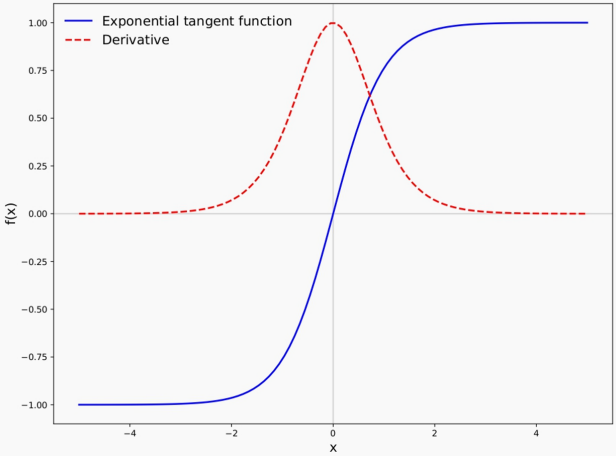
Another issue is that the sigmoid transformation is not zero-centered

$$\frac{\partial \mathcal{L}(\beta)}{\partial \beta_j^{(2)}} = \delta^{(2)} \frac{\partial z^{(2)}}{\partial \beta_j^{(2)}} = \delta^{(2)} x_j^{(1)} \tag{2}$$

where $x_j^{(1)}$ is always positive

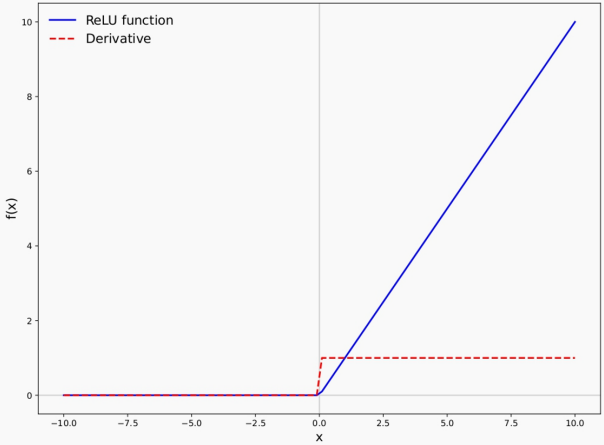
- This causes the update for all the parameters of a layer to be either positive or negative
- This restricts the optimization and takes longer to converge, as the path has a zigzag shape

Exponential Tangent



Another common activation is the exponential tangent function, which is zero-centered. The restricted range, while larger than sigmoid, can still cause units to saturate, and the function remains computationally expensive.

ReLU



The ReLU function does not saturate in the positive region, is computationally efficient, and allows for faster convergence (Krizhevsky et al. 2012). This function allows training larger networks and is widely used as the default activation in modern networks.

ReLU

A large negative constant can cause z_{iu} to be negative and $\sigma(z_{iu}) = 0$, so that the parameters are not updated

- The constant is not updated (i.e., does not change with the mini-batch), so the **unit can effectively "die"**
- This may happen with large learning rates, and the constant should be initialized to a positive value
- Extensions to ReLU avoid this problem at the cost of additional parameters, e.g., PReLU (He et al. 2015)

Better Initialization

Better Initialization

Randomly initialized parameter values have a significant impact on the optimization procedure

- They define the starting position on the loss function graph and the minima optimization can reach
- They interact with the activation functions and can cause the gradients to vanish, i.e., $\sigma^{(l)}(x^{(l-1)} \cdot \beta_u^{(l)})$
- We derive a principled way to initialize the parameters for effective optimization (Glorot and Bengio 2010)

Intuition

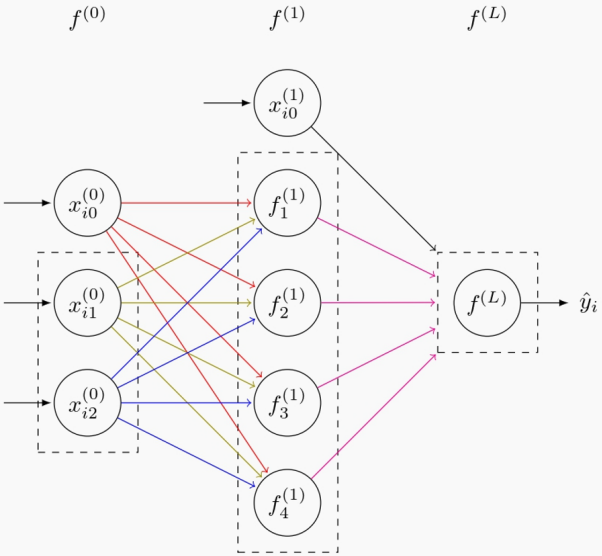


Figure: Initialization with constant values

The contribution of each colored parameter to the loss (i.e., partial derivative) will be the same, so they will receive the same update (i.e., symmetry problem). Their value changes during optimization, but will remain tied, so the model becomes much less flexible as hidden units compute the same features.

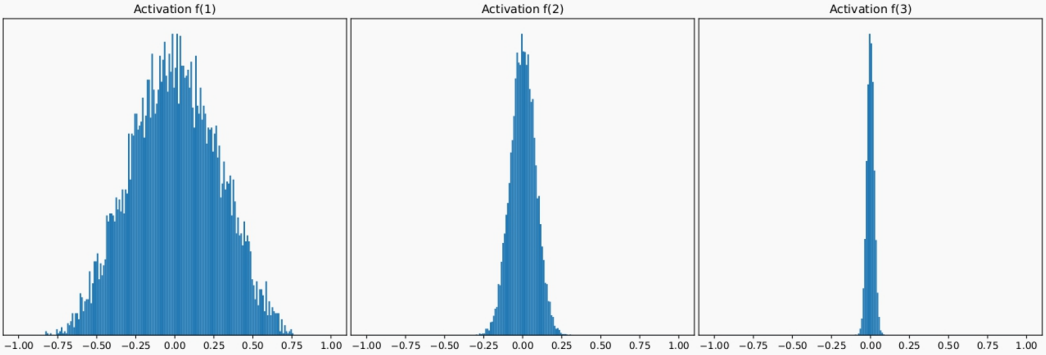
Better Initialization

Random Initialisations

```
# Modules
import numpy as np

# Initialises data
X = np.random.normal(size=(4, 100, 32))

# Initialises parameters (small, large, normal, choose one)
B = np.random.uniform(low=-0.05, high=0.05, size=(4, 32, 32))
B = np.random.uniform(low=-0.5, high=0.5, size=(4, 32, 32))
B = np.random.normal(size=(4, 32, 32)) / (np.sqrt(32))

# Forward propagation (replaces Xl, l=1,...,L)
for l in range(1, 4):
X[l] = np.tanh(X[l-1] @ B[l])
```

Intuition

Figure: Initialization with small random values



Distribution of unit's output $x_u^{(l)}$ for layers $f^{(l)}$, $l = 1, \ldots, 3$. Units use the tanh activation, and the initial parameters are drawn from the uniform distribution $[-0.05, +0.05]$.

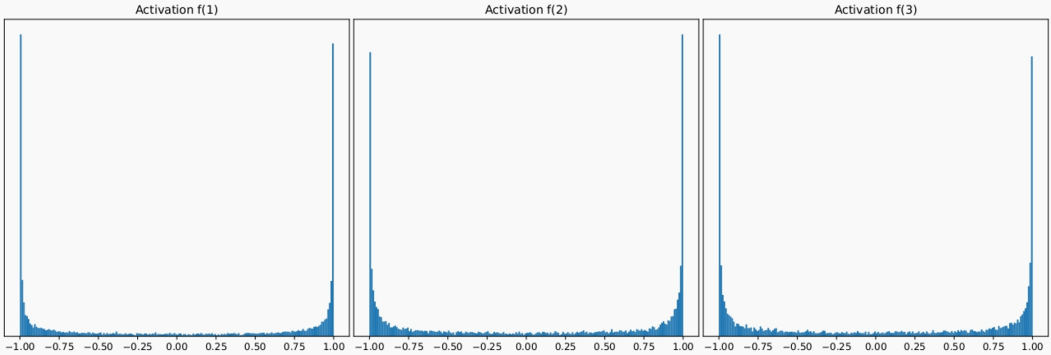
Intuition

The units' output $x_u^{(l)}$ converge toward the center of the activation (i.e. 0 for tanh) with every hidden layer

- There is little variance in the units' output, and their contribution to the loss is close to 0
- The gradients are also close to 0, as they are the product of many partial derivatives that are close to 0
- The parameters of the early layers are not updated effectively (i.e., vanishing gradient problem)

Intuition

Figure: Initialization with larger random values



Distribution of unit's output $x_u^{(l)}$ for layers $f^{(l)}$, $l = 1, \ldots, 3$. Units use the tanh activation, and the initial parameters are drawn from the uniform distribution $[-0.5, +0.5]$.

Intuition

Individual parameters are not large, but given enough hidden input units, the dot product $z_u^{(l)} = x^{(l-1)} \cdot \beta_u^{(l)}$ is large

- Large positive of negative $z_u^{(l)}$ passed to the exponential tanh function are close to -1 or 1 (i.e., saturation)
- The partial derivative of the activation function in these areas is close to 0 (i.e., vanishing gradients)
- We need a more principled way of initializing the parameters by **controlling variance in the units' output** (Glorot and Bengio 2010)

Glorot

Consider the variance for the z of a single unit, layer and unit indexes are dropped for clarity

$$V(z) = V\left(\sum_{i=1}^{n} \beta x_i\right) = \sum_{i=1}^{n} V\left(\beta x_i\right)$$

$$V(z) = \sum_{i=1}^{n} \left[\underline{E(\beta)}^2 V(x_i) + \underline{E(x_i)}^2 V(\beta) + V(x_i)V(\beta)\right]$$

$$V(z) = \underline{\sum_{i=1}^{n}} V(x_i)V(\beta) = \left(\underline{n} V(\beta)\right) V(x) \tag{3}$$

The variance[3] of the output is the variance of the input scaled by $nV(\beta)$, we need an initialization such that $nV(\beta) = 1$

[3] The normalized inputs and initialized parameters have mean 0 and the same variance (i.e. $V(x_i) = V(x),\ \forall i$)

Glorot

Adding layer indexes and unit, and ignoring activation

$$V(z^{(1)}) = \left[nV\left(\beta^{(1)}\right)\right] V(x)$$

When $nV(\beta) \gg 1$, then $V(z)$ is large and when $nV(\beta) \to 0$ this variance is low. For the subsequent unit

$$\begin{aligned} V(z^{(2)}) &= \left[nV\left(\beta^{(2)}\right)\right] V(z^{(1)}) \\ V(z^{(2)}) &= \underline{\left[nV\left(\beta^{(2)}\right)\right] \left[nV\left(\beta^{(1)}\right)\right]} V(x) \\ V(z^{(2)}) &= \underline{[nV(\beta)]^2} V(x) \end{aligned}$$

Assuming that all the $\beta^{(l)}$ have the same variance i.e. $V\left(\beta^{(l)}\right) = V(\beta)$

Glorot

We need that $nV(\beta) = 1$, under this condition, the units will not saturate not vanish. A solution is

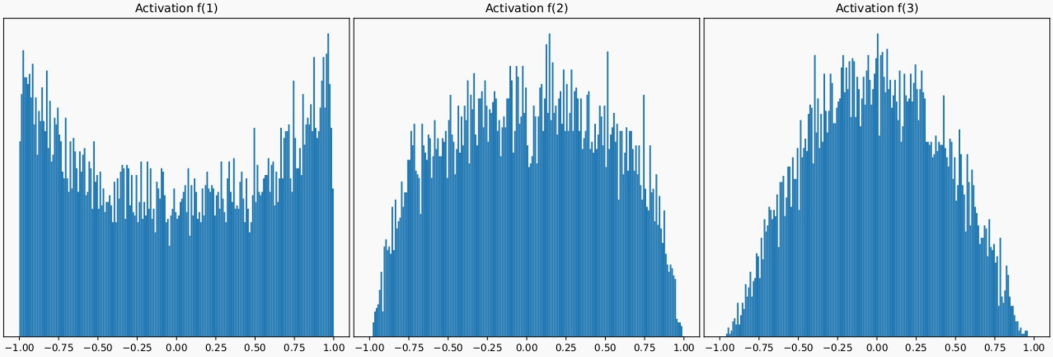
$$nV(\beta) = nV\left(\frac{z}{\sqrt{n}}\right) = n\frac{1}{n}V(z) = 1 \tag{4}$$

Using the fact that $V(az) = a^2 V(z)$

- I have the draw the parameters from a given distribution so that this condition holds
- Under this condition, the variance will neither blow up nor shrink

Glorot

Figure: Normal initialization



For the ReLU function, this initialization does not work (He et al. 2015). See Narkhede et al. (2022) for a review.

Batch Normalization

The input variables x_u are usually standardized to have zero mean and unit standard deviation

$$x_{iu}^s \Leftarrow \frac{x_{iu} - \bar{x}_u}{\sqrt{\sigma_u^2 + \varepsilon}}$$

$$\bar{x}_u = \frac{1}{n} \sum_{i=1}^{n} x_{iu}$$

$$\sigma_u^2 = \frac{1}{n} \sum_{i=1}^{n} (x_{iu} - \bar{x}_u)^2$$

where ε is a small number to avoid zero division, the layer index is dropped for simplicity

Batch Normalization

Batch normalization (Ioffe and Szegedy 2015) is a layer that standardizes the input to the next layer

- This transformation is applied to each batch, as they change during optimization (i.e., different distributions)
- At each optimization iteration, x_{iu} is standardized using the current batch's mean and standard deviation
- Batch normalization controls the first two moments of a unit's x_{iu} input distribution for a given batch

Batch Normalization

In some applications, having different input distributions can be useful for prediction (e.g., separating classes)

$$x_{iu}^{bn} \Leftarrow \theta_{0u} + \theta_{1u} \cdot x_{iu}^{s}$$

where θ_{0u} and θ_{1u} are the shift and scaling factors for the unit's output distribution, respectively

- This operation is differentiable, so the model can estimate the batch normalization parameters (i.e., 4 for each unit)
- During predictions, we aggregate the mean and standard deviation using an exponentially weighted moving average over the training mini-batches[4]

[4] Layers like normalization and dropout behave differently at train and test time. Make sure the model is in the correct mode.

Batch Normalization

Batch normalization smooths the loss landscape (Santurkar et al. 2018), enabling faster and more robust convergence[5]

- Units are less sensitive to the distribution of their input from previous layers (i.e., internal covariate shift)
- Better used before activation (i.e. z_{iu}) when the transformation produces a non-Gaussian distribution
- In this case, the constant term becomes redundant as batch-normalization controls the shift factor

[5] Other types of normalization are used, e.g., layer normalization in recurrent models or transformers.

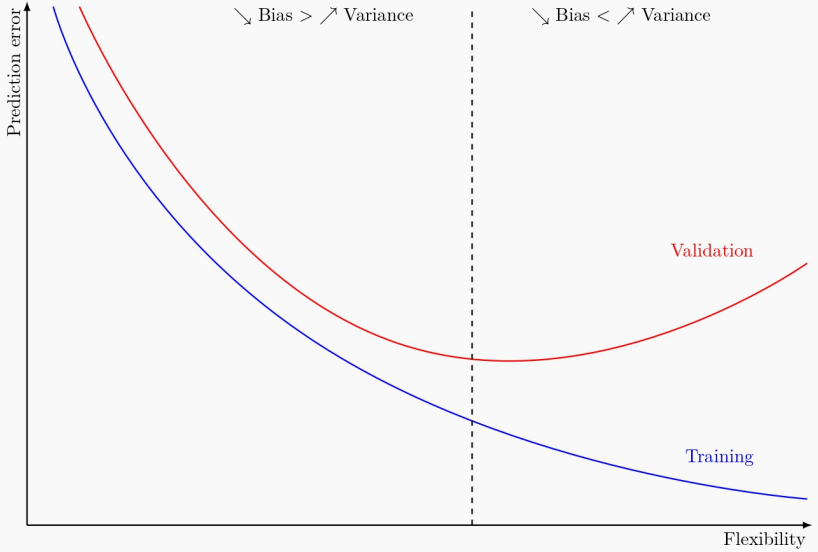
Better Regularization

Early-Stopping

Early stopping uses another random sample called validation to monitor the model's generalization during training

- The validation error is computed after each training epoch, and monitored with a patience parameter
- When the model starts overfitting, the training error decreases while the validation error increases
- The test sample is separate from the validation sample because the latter is used to select the parameters

Early-Stopping



Early-Stopping

Early stopping reduces the number of optimization iterations and constrains the maximum size of the parameter updates

- The available data is commonly partitioned into 70% for training, 15% for validation and 15% for testing
- A disadvantage of early stopping is that less of the available data is being used to estimate the parameters
- This regularization technique can be used with other regularization procedures (e.g., L_2, or dropout)

Dropout

Bootstrap aggregation can be used to reduce the variance of a statistical model (errors are not perfectly correlated)

- Estimating different networks (i.e., decorrelation) on different subsamples is computationally demanding
- **Dropout** (Nitish 2014) is a technique to train an ensemble of networks at no additional computational cost
- This is done by randomly dropping a sample of input or hidden units at each optimization iteration

Dropout

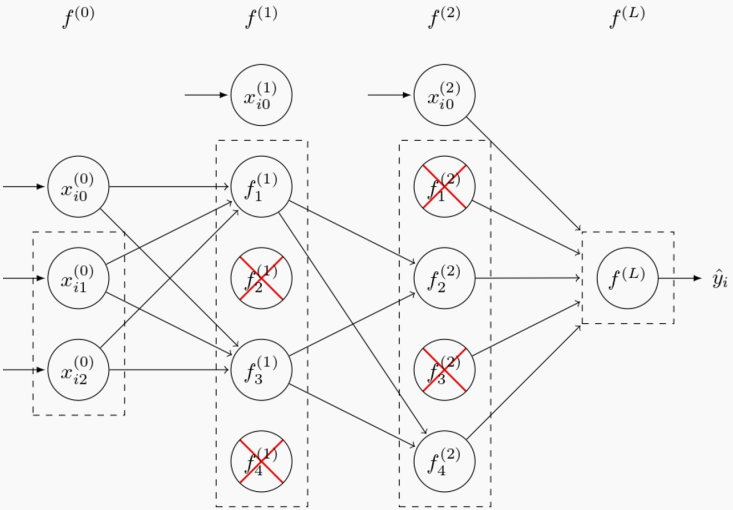


Figure: Dropout Regularization 1

Each unit is associated with a probability of being kept (e.g., $p = 50\%$). The parameters are shared and we sample a different sub-network across optimization iterations.

Dropout

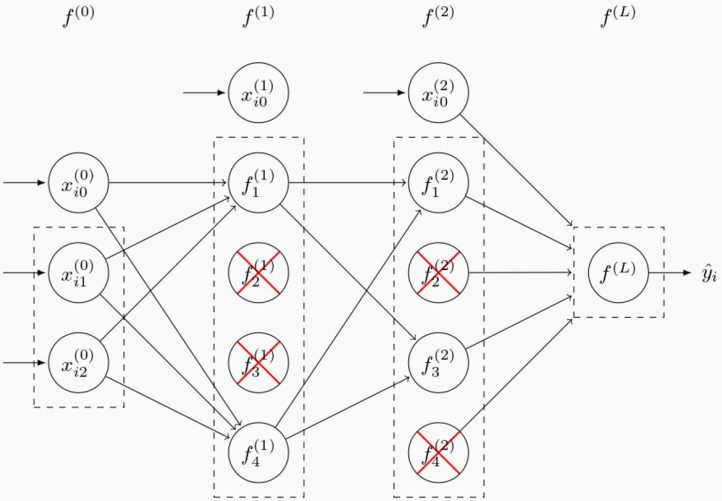


Figure: Dropout Regularization 2

Only the kept units are updated during back-propagation. Some sub-networks may never be sampled, which is fine since the sub-network parameters are shared.

Dropout

We study the inverted dropout implementation that specifies a probability of the unit being kept (i.e., simpler at test time)

$$X_D^{(l)} = \frac{X^{(l)} \odot \left(D^{(l)} < p\right)}{p} \tag{5}$$

where $D^{(l)}$ is a random array of values between 0 and 1 from the uniform distribution and p is the keep probability

- $D^{(l)}$ has the same dimensions as $X^{(l)}$, for every observation in the batch, different units are dropped
- The expected value of $X_D^{(l)}$ decreases by $(1 - p)$ the dropout rate so we divide by the keep probability

Dropout

Dropout reduces the units' co-adaptation and prevents them from becoming inactive by depending on others

- Since subsequent units cannot rely on a particular input, the parameters will be more evenly distributed
- This effectively shrinks larger parameter values, which has a similar effect to other regularization techniques
- Dropout encourages some redundancy and the detection of diverse patterns, making the model more robust

Dropout

Dropout should be used when the model overfits, in layers with many parameters (e.g., computer vision)

- Dropout is deactivated for predictions, and the rescaling ensures that the computations remain valid
- Dropout can also be used on the input layer to add noise to the training data, when the input is redundant
- The optimization path is less smooth as the shape of the loss function changes with every iteration

Summary

Summary

We covered numerous developments in the deep learning literature that allowed us to **optimize large networks**

- Better optimization procedures (e.g. mini-batches, batch-norm., Adam) and activation functions (e.g. ReLU)
- Better initialization strategies (e.g., Golorot, He), regularization techniques (e.g., early stopping, dropout)
- These elements **interact in complex ways**, and we need an intuitive understanding of their mechanisms and impact