Deep Learning

MLSS 2019

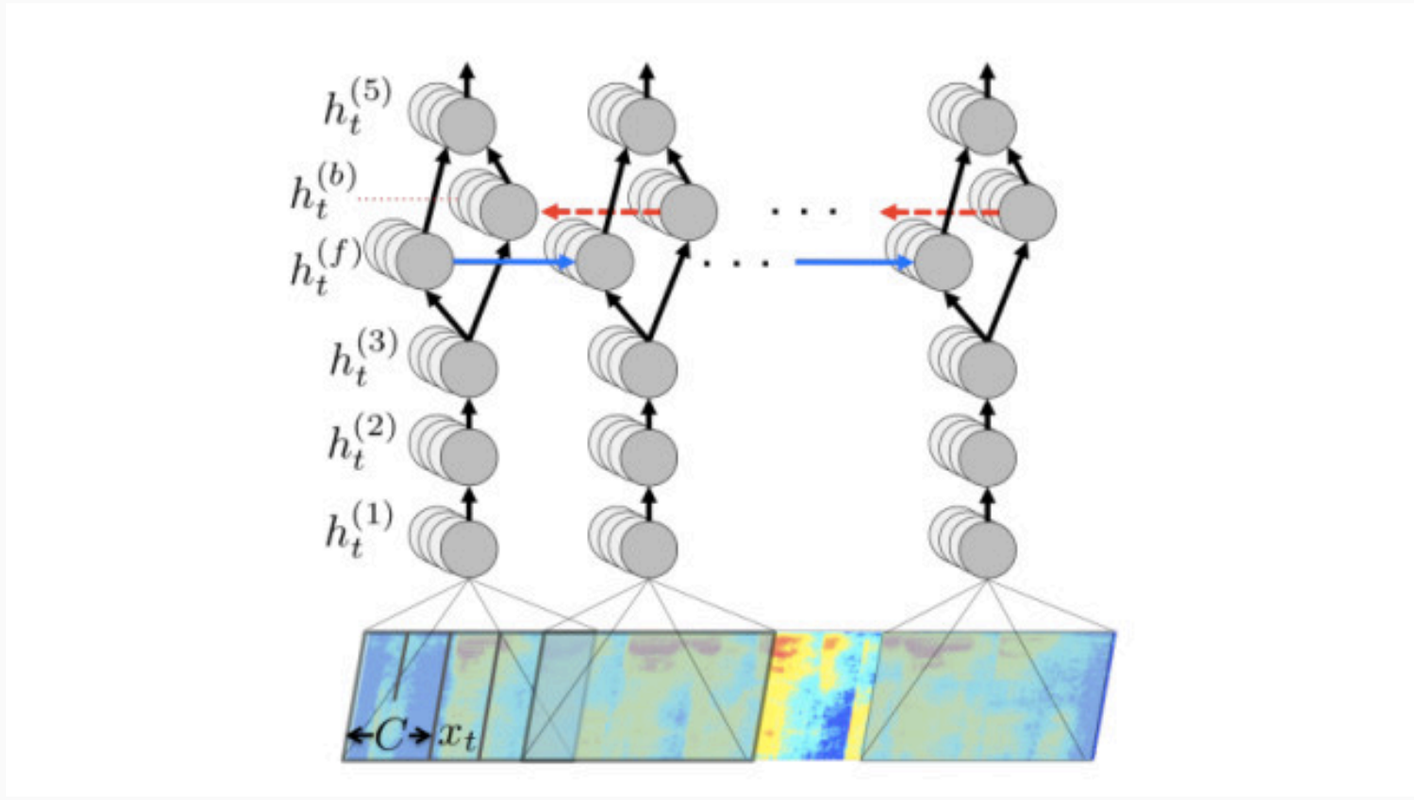
Kevin Webster, Pierre Harvey Richemond

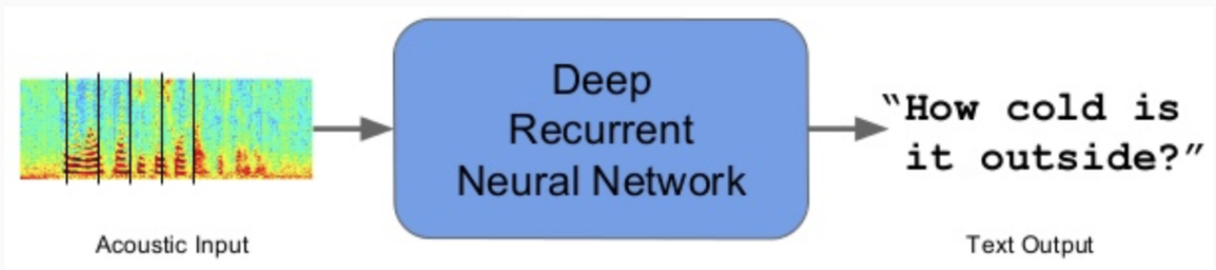
July 17th, 2019

Outline

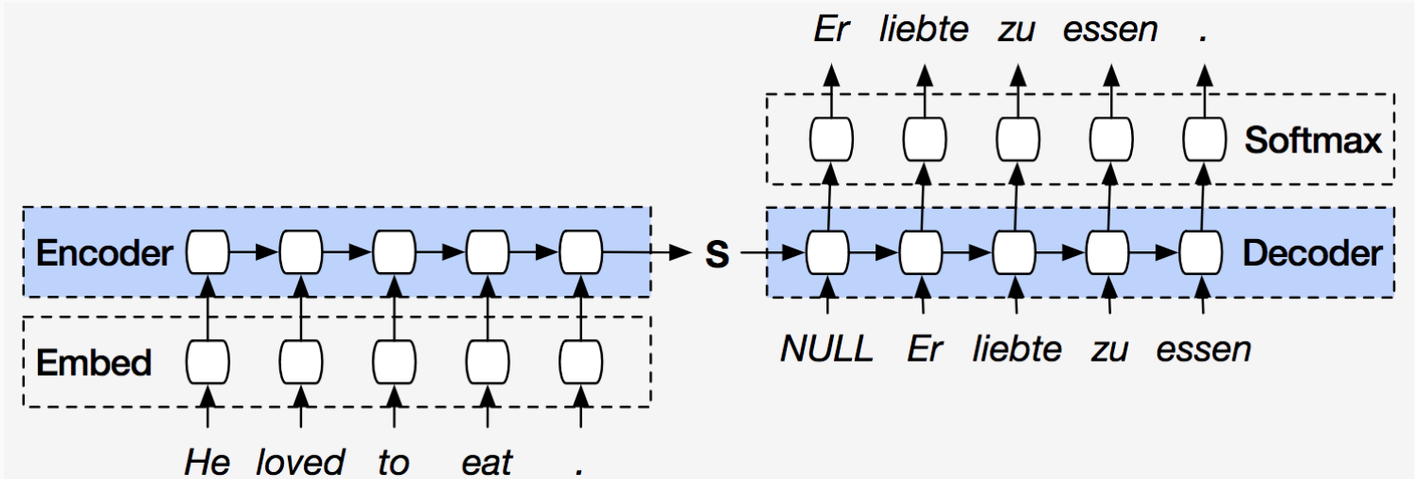
Outline

Speech recognition





Machine translation



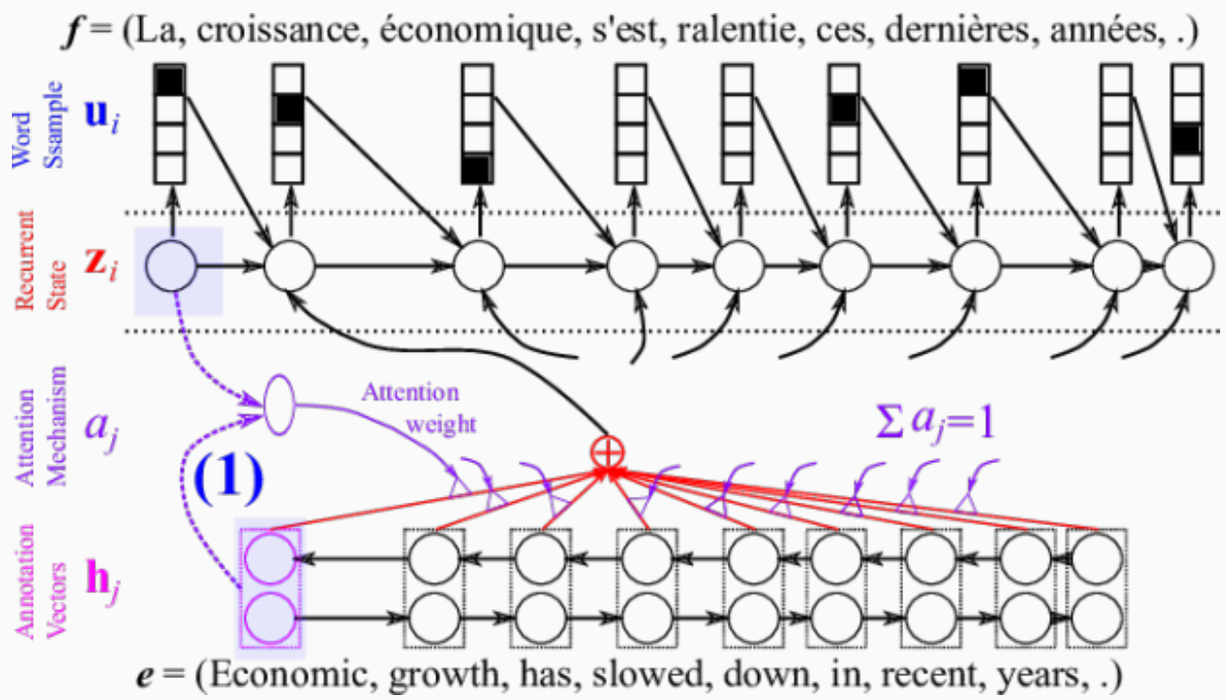
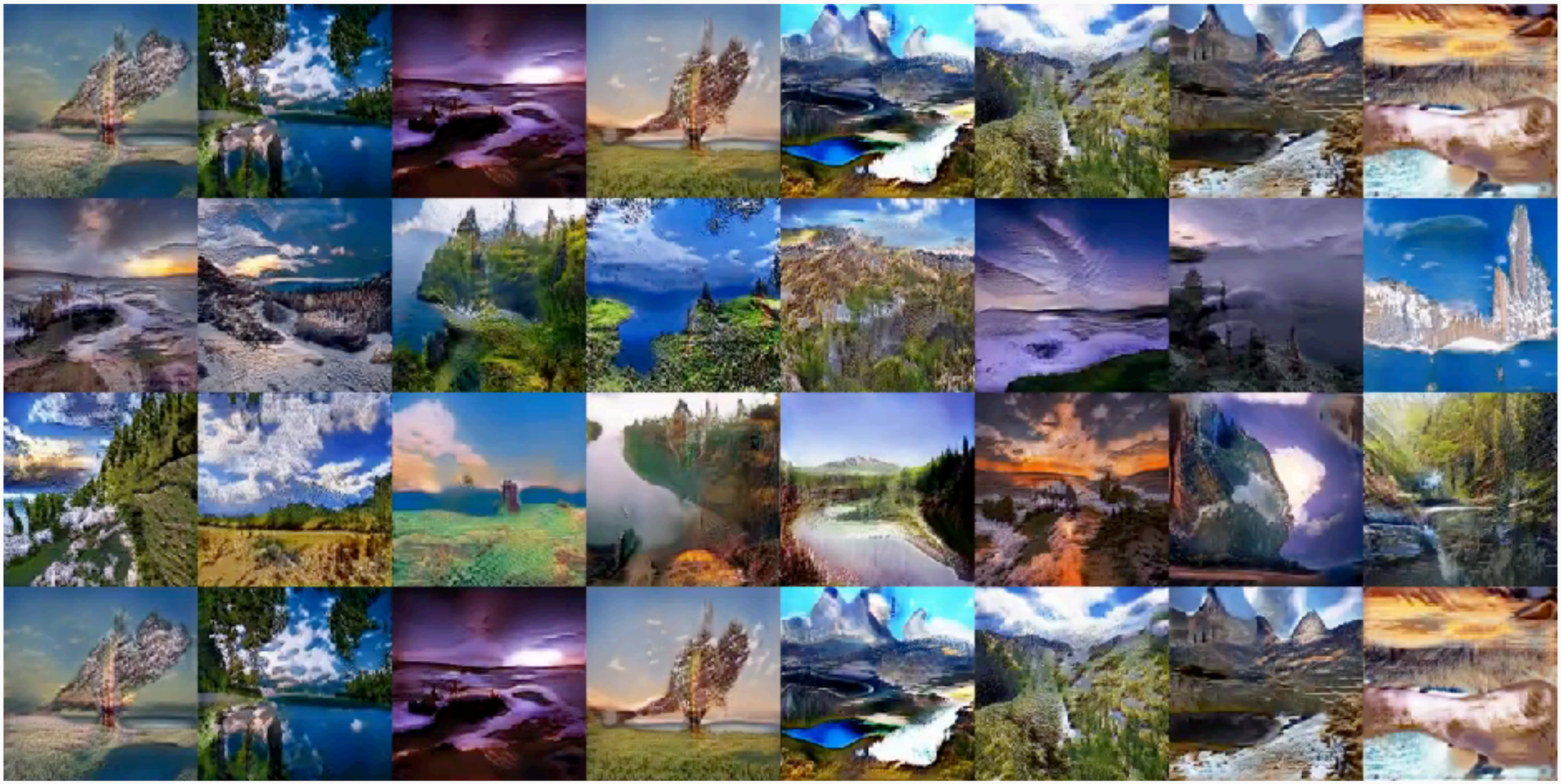


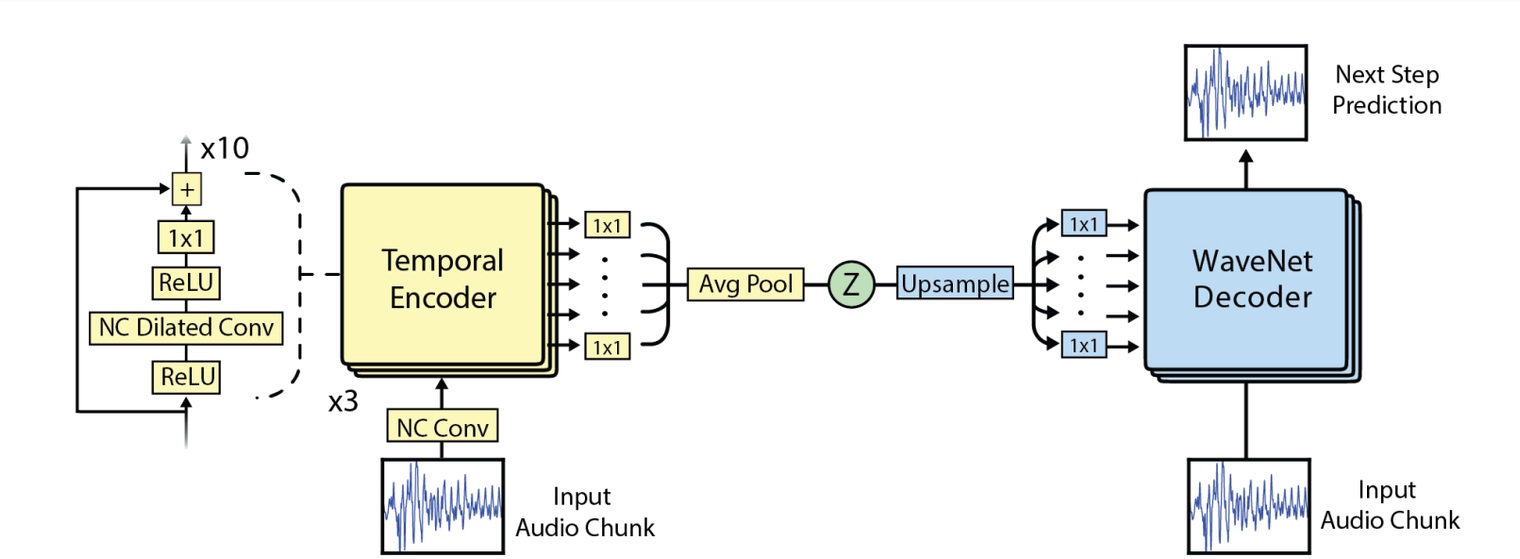
Image generation

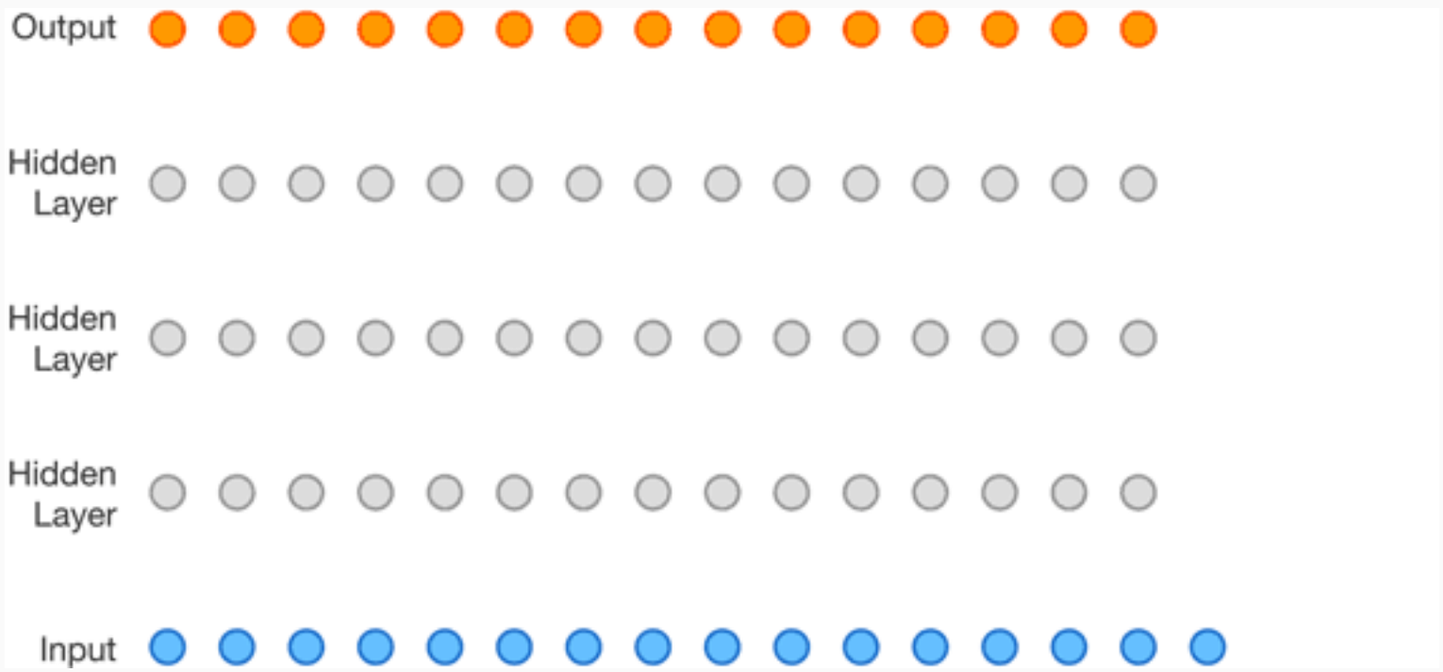
Image generation



Deep Learning applications

Audio synthesis



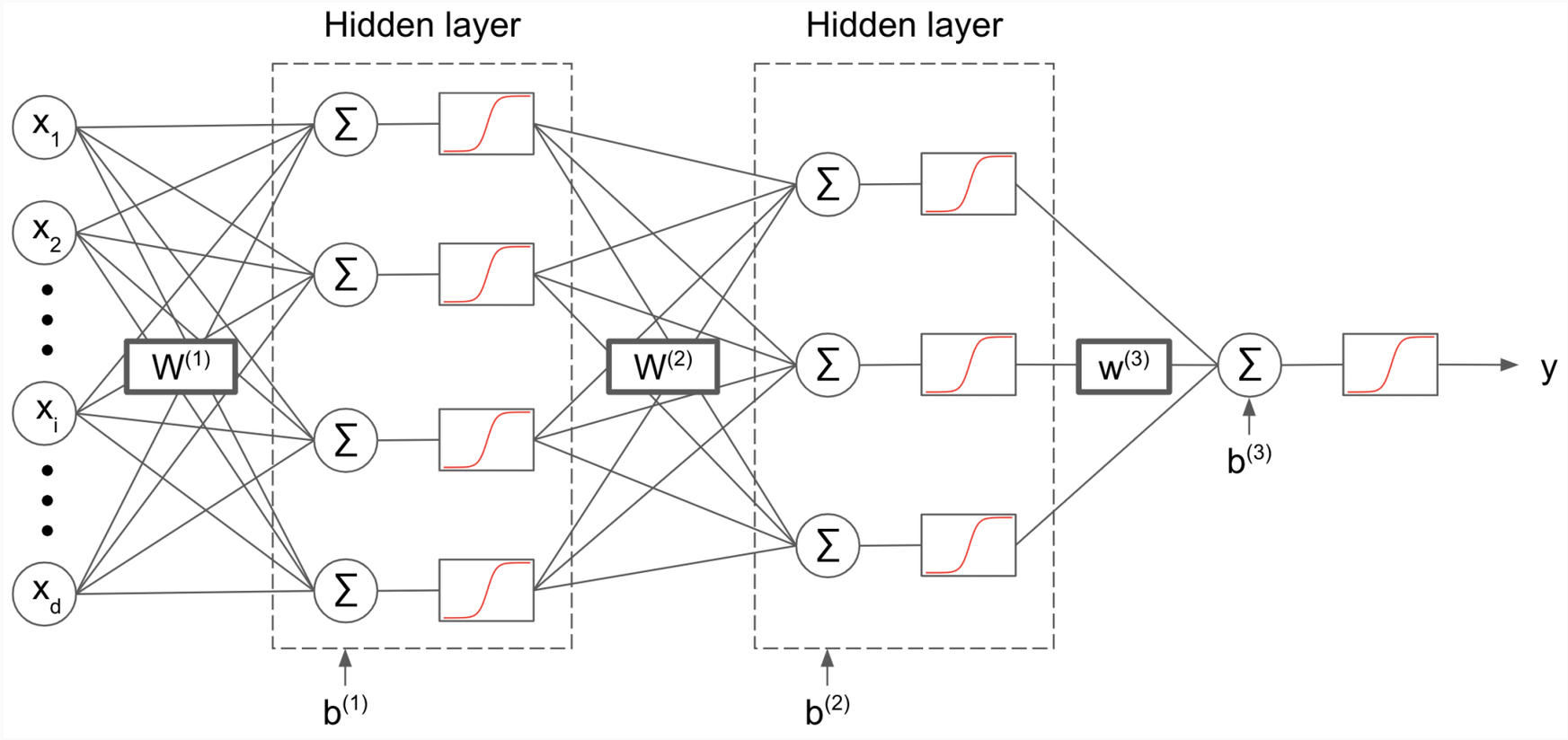


Deep Learning models

- Neural network models can be seen as a modelling approach where the number of basis functions are fixed, but are themselves parameterised
- The parameters are adapted during the training phase
- We will begin by considering neural network models for classification and regression

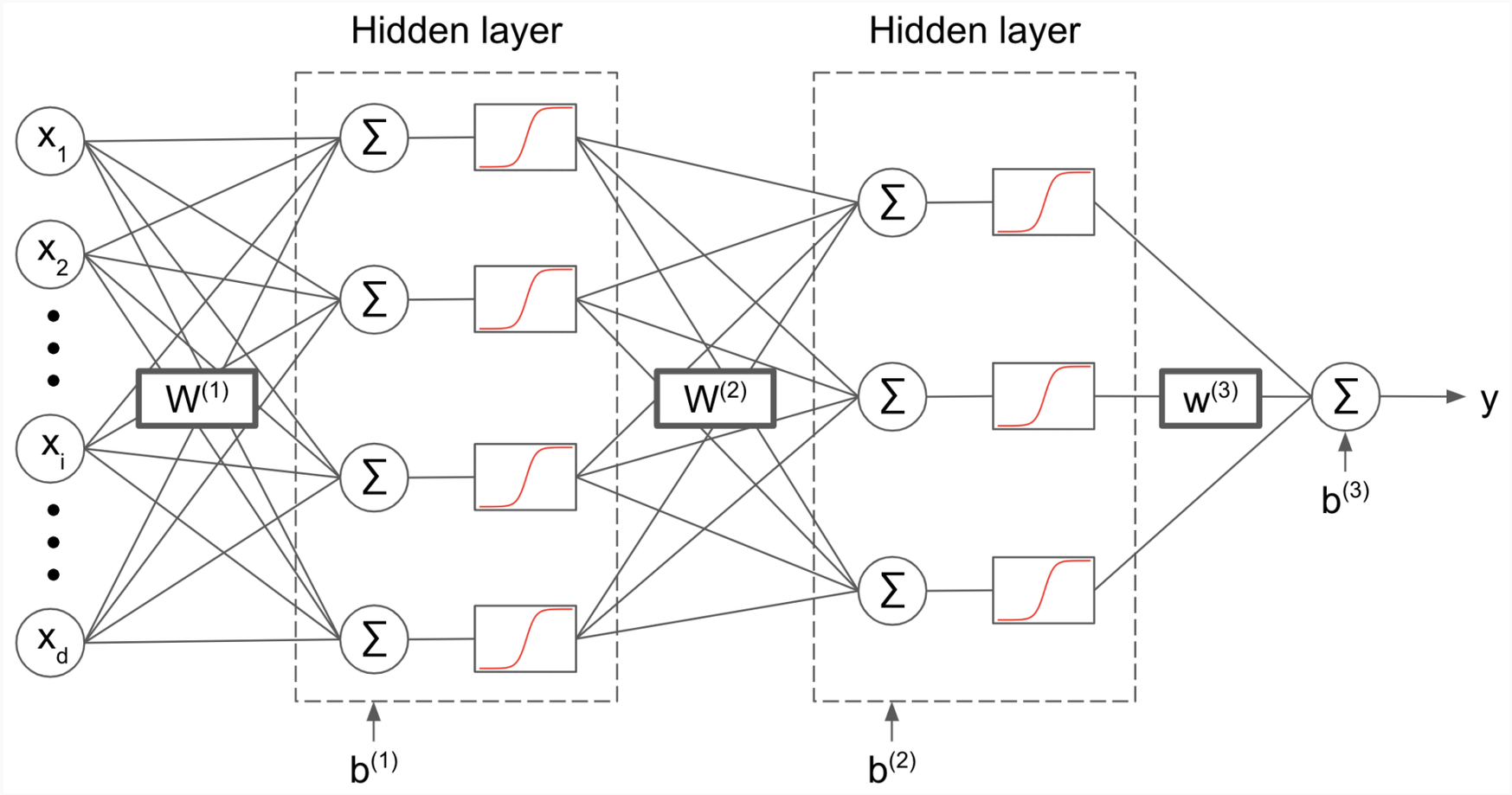
- A common type of neural network is called a **feedforward network** or a **multilayer perceptron** (MLP)
- The MLP is constructed by stacking hidden layers of logistic regression functions, terminating in an output prediction layer
- The term *multilayer perceptron* is a misleading name, since it uses continuous nonlinearities like the sigmoid function (as in logistic regression) instead of a discontinuous step function activation (as in the perceptron)

Feedforward network / MLP: binary classification



- The MLP constructed above consists of two hidden layers of neurons, each of which has the form of a logistic regression model
- The output y can be thought of as a logistic regression model where the basis functions are determined by the neurons in the last hidden layer

Feedforward network / MLP: regression



- An MLP can also be used to solve regression problems. The hidden layers are constructed in the same way
- The only difference is in the output layer, which can be thought of as a linear combination of basis functions given by the neurons in the last hidden layer

Deep learning

- The ‘deep’ in deep learning refers to the number of hidden layers in the network
- In general, greater depth in networks allow for a richer class of approximating functions
- Intuition is that deep networks allow the model to build hierarchies of concepts
- Concepts are built on top of each other in layers
- This reduces the need for hand-engineered features (basis functions)
- Deep learning can be seen as part of *representation learning*, which aims to discover the best representations or features of the data

Deep learning frameworks

Caffe

DL4J
DEEPLEARNING4J

The Microsoft Cognitive Toolkit
A free, easy-to-use, open-source, commercial-grade toolkit that trains deep learning algorithms to learn like the human brain.

mxnet

PYTORCH

TensorFlow

BigDL

Caffe2

torch

Chainer

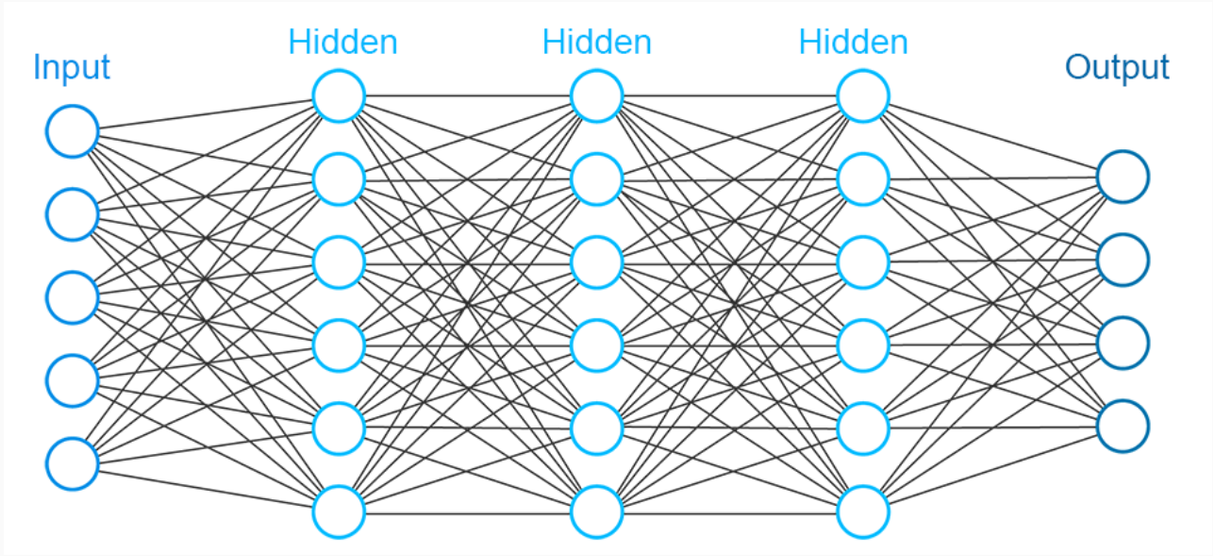
theano

Deep learning

- There is a great deal of flexibility when it comes to constructing deep learning models
- We will look at the functional form of neural networks, including choice of nonlinear activation functions
- The parameters of the network are tuned by optimising the data likelihood
- The representational power of neural networks comes at the price of a non-convex loss function, and a potentially large set of parameters
- Neural networks are trained using gradient-based methods
- We will look at the process of tuning network parameters, especially the *backpropagation* algorithm

Outline

Feedforward network / multilayer perceptron



Neural network with L layers $h_1, \ldots, h_L$, where $h_i \in \mathbb{R}^{n_i}$. Input $\mathbf{x} = h_1$ and output $\mathbf{y} = h_L$. For $i = 1, \ldots, L-1$, we define

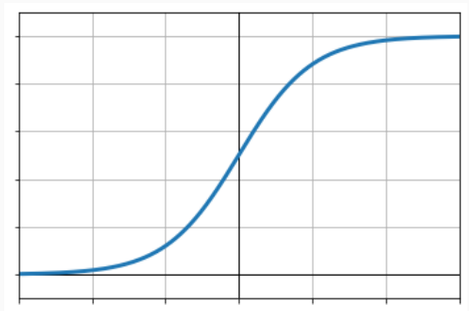
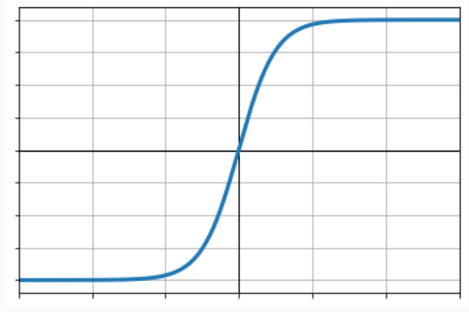
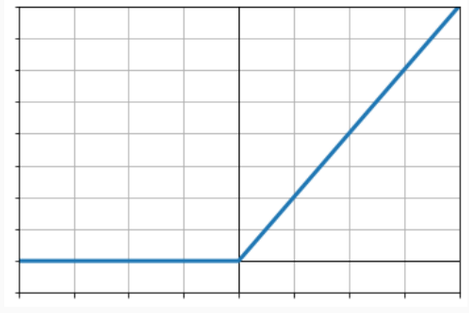
$$\hat{h}_{i+1} = W^{(i)} h_i + b^{(i)}, \qquad \text{(pre-activation)}$$

where $W^{(i)} \in \mathbb{R}^{n_{i+1} \times n_i}$ and $b^{(i)} \in \mathbb{R}^{n_{i+1}}$.

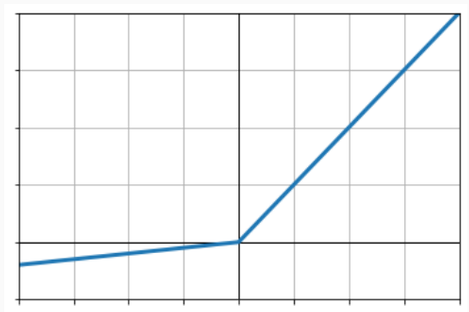
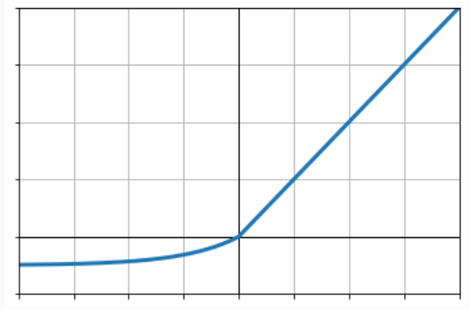
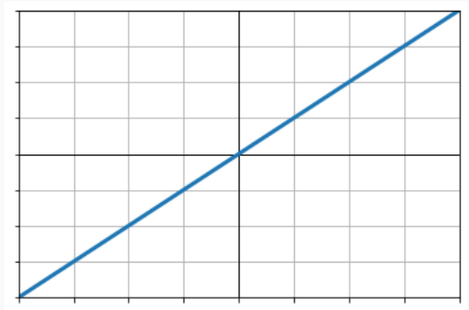
$$h_{i+1} = \sigma(\hat{h}_{i+1}), \qquad \text{(post-activation)}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is an **activation function** that is applied element-wise.

Activation functions

Activation function	Plot	Equation
sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
tanh		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLU		$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$

Activation functions

Activation function	Plot	Equation
leaky ReLU		$f(x) = \begin{cases} \epsilon x, & x < 0 \\ x, & x \geq 0 \end{cases}$
ELU		$f(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$
identity		$f(x) = x$

Output layers

Frequently used output activations / layers:

- **Sigmoid output layer** is useful to predict probabilities as the range is in $(0, 1)$.
- **Softplus output layer** is useful to predict e.g. Gaussian variance parameters, as the range is $(0, \infty)$:

$$f(x) = \ln(1 + e^x)$$

- **Softmax output layer** is often used to predict parameters of a categorical distribution:

$$P(\text{Category } j) = \frac{e^{z_j}}{\sum_{k=1}^{M} e^{z_k}},$$

where z_k $(k \in \{1, \ldots, M\})$ are the pre-activations.

Loss functions

Loss functions used in deep learning are often attempting to maximise log-likelihood. Typical per-example loss functions are:

- **Cross-entropy loss**. Frequently used in classification, with M mutually exclusive classes:

$$H_{\hat{\mathbf{y}}}(\mathbf{y}) := -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\hat{y}_i^{(j)}\log(y_i^{(j)})$$

- **Mean squared error**. Frequently used in regression tasks, e.g. in predicting a t-dimensional output:

$$S := \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{t}(y_i^{(j)} - \hat{y}_i^{(j)})^2$$

Other loss functions are used for particular network frameworks such as the VAE or GAN.

Regularisation

Regularisation is important to avoid overfitting. There are several approaches to regularisation in deep learning:

- Model complexity
- Weight decay
- Patience/early stopping
- Dropout
- Weight sharing
- Ensemble predictions
- Dataset augmentation
- Noise robustness
- Multitask learning
- Adversarial training

Stochastic gradient descent

Most Deep Learning models are trained with (a variant of) stochastic gradient descent.

The cost function usually decomposes as a sum over the training examples in the training set:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ are the parameters of the model, $(\mathbf{x}_i, y_i)_{i=1}^{N}$ is the (labelled) training set, and L is the per-example loss.

Stochastic gradient descent

Gradient descent involves computing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_i, y_i, \boldsymbol{\theta}),$$

which can be prohibitively expensive for large datasets.

Stochastic gradient descent approximates the expected gradient by sampling a **minibatch** of data examples $(\mathbf{x}_{n_i}, y_{n_i})_{i=1}^{N'}$, $N' \ll N$.

The estimated gradient is then

$$\mathbf{g} = \frac{1}{N'} \sum_{i=1}^{N'} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}_{n_i}, y_{n_i}, \boldsymbol{\theta}),$$

Stochastic gradient descent

Stochastic gradient descent then uses the estimated gradient to adjust the parameters:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g},$$

where ϵ is the learning rate.

- SGD makes efficient use of the dataset when training deep learning models
- The cost per SGD update does not depend on the training size N
- Provides a scalable way of training nonlinear models on large datasets

Outline

Maximum likelihood

Just as with the other parametric models, we need to estimate the parameters of the neural network. The parameters are the weight matrices $W^{(i)} \in \mathbb{R}^{n_{i+1} \times n_i}$ and biases $b^{(i)} \in \mathbb{R}^{n_{i+1}}$ for $i = 1, \ldots, L-1$.

Neural networks are often trained using maximum likelihood. Our aim is to maximise the likelihood function over the whole dataset:

$$\begin{aligned} \boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}|\mathbf{x}, \mathbf{y}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p(y_i|x_i, \boldsymbol{\theta}) \end{aligned}$$

where we have assumed the data to be independent and identically distributed (i.i.d.).

Maximum likelihood: regression

Consider first the regression setting. We are given a dataset consisting of N input points

$$\mathbf{x} = (x_1, x_2, \ldots, x_N), \qquad x_i \in \mathbb{R}^d \quad \forall i$$

and N corresponding output values

$$\mathbf{y} = (y_1, y_2, \ldots, y_N), \qquad y_i \in \mathbb{R} \quad \forall i$$

We wish to find a parametric function $f(x, \boldsymbol{\theta})$ that models the relationship between $\mathbf{x}$ and $\mathbf{y}$.

Maximum likelihood: regression

- We assume that the data is generated according to the following model

$$y = f(x, \boldsymbol{\theta}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2),$$

  where σ^2 is a hyperparameter
- Our assumption is that the data is contaminated by Gaussian noise, just as in a linear regression model
- However, now the function $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto \mathbb{R}$ is our neural network, where p is the total number of model parameters (weights and biases)
- Note that in this case, the final output layer of the network consists of a single neuron with no nonlinear activation function
- The probability $p(y_i|x_i, \boldsymbol{\theta})$ is given by the Gaussian pdf with mean $f(x_i, \boldsymbol{\theta})$ and variance σ^2 for each data example (x_i, y_i).

Maximum likelihood: regression

As usual, it is convenient to work with the negative of the log-likelihood function, as the arg min of the negative log-likelihood is the same as the arg max of the likelihood function.

$$\begin{aligned}
\boldsymbol{\theta}_{ML} &= \arg\min_{\boldsymbol{\theta}} -\log \mathcal{L}(\boldsymbol{\theta}|\mathbf{x}, \mathbf{y}) \\
&= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} -\log p(y_i|x_i, \boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - f(x_i, \boldsymbol{\theta}))^2 + \text{const.}
\end{aligned}$$

where the constant is independent of the model parameters $\boldsymbol{\theta}$. We have derived the **loss function** (MSE) to be optimised:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i, \boldsymbol{\theta}))^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \quad (\hat{y}_i := f(x_i, \boldsymbol{\theta}))$$

Maximum likelihood: binary classification

We treat the classification in a similar way. Our dataset again consists of N input points

$$\mathbf{x} = (x_1, x_2, \ldots, x_N), \qquad x_i \in \mathbb{R}^d \quad \forall i$$

and N corresponding output values

$$\mathbf{y} = (y_1, y_2, \ldots, y_N), \qquad y_i \in \{0, 1\} \quad \forall i$$

where $y_i = 1$ if $x_i \in \mathcal{C}_1$ and vice versa.

In the above setting, our neural network output layer consists of a single neuron with a logistic sigmoid activation function. This ensures that the network output is interpretable as a probability: $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto (0, 1)$.

As in the case of logistic regression, we use the network function $f(x, \boldsymbol{\theta})$ to model the probability $p(\mathcal{C}_1|x)$, and $p(\mathcal{C}_2|x) = 1 - p(\mathcal{C}_1|x)$.

Maximum likelihood: binary classification

Again, we seek the parameters that minimise the negative log-likelihood:

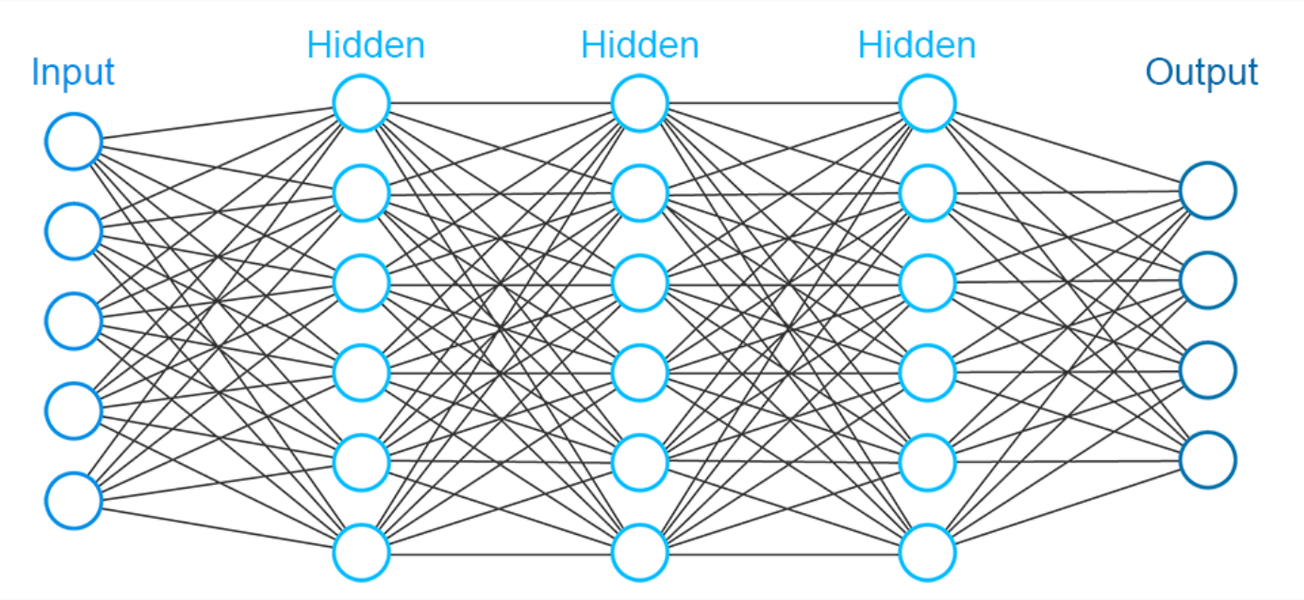
$$\begin{aligned}
\boldsymbol{\theta}_{ML} &= \arg\min_{\boldsymbol{\theta}} -\log \mathcal{L}(\boldsymbol{\theta}|\mathbf{x}, \mathbf{y}) \\
&= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} -\log p(y_i|x_i, \boldsymbol{\theta}) \\
&= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} -y_i \log f(x_i, \boldsymbol{\theta}) - (1 - y_i) \log(1 - f(x_i, \boldsymbol{\theta}))
\end{aligned}$$

We have derived the **loss function** (binary cross entropy) to be optimised:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i)$$

where we denote the model prediction $\hat{y}_i := f(x_i, \boldsymbol{\theta}) = p(\mathcal{C}_1|x_i)$.

Multiple target dimensions: regression



The previous derivations can be extended in a straightforward way to a multi-dimensional target variable.

In the case of regression, we consider inputs $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ with $x_i \in \mathbb{R}^d$ and outputs $\mathbf{y} = (y_1, y_2, \ldots, y_N)$ with $y_i \in \mathbb{R}^t$.

We denote the j-th dimension of y_i as $y_i^{(j)}$. Our network is now a mapping $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto \mathbb{R}^t$ with t (linear) output neurons.

Multiple target dimensions: regression

We make the similar modelling assumption that the data is generated according to the following model

$$y = f(x, \boldsymbol{\theta}) + \epsilon, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}_t)$$

Similar arguments follow to show that the maximum likelihood solution is given by minimising the following MSE loss function:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{t} (y_i^{(j)} - \hat{y}_i^{(j)})^2$$

where we denote the model prediction $\hat{y}_i = f(x_i, \boldsymbol{\theta}) \in \mathbb{R}^t$.

Multiple target dimensions: classification

For classifications, we may consider two different settings for multiple target dimensions.

In the simpler case, we have inputs $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ with $x_i \in \mathbb{R}^d$ and M binary output labels $\mathbf{y} = (y_1, y_2, \ldots, y_N)$ with $y_i \in \{0, 1\}$. In this case we have M separate binary classifications to perform.

In this setting, our neural network output layer consists of M neurons with a logistic sigmoid activation function, and $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto (0, 1)^M$.

With similar arguments to before, we derive the loss function

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} -y_i^{(j)} \log \hat{y}_i^{(j)} - (1 - y_i^{(j)}) \log(1 - \hat{y}_i^{(j)})$$

Multiple target dimensions: classification

For multiclass classification, we have inputs $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ with $x_i \in \mathbb{R}^d$ and outputs $\mathbf{y} = (y_1, y_2, \ldots, y_N)$ with each $y_i \in \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$.

In this case, we typically encode the data so that each class $\mathcal{C}_j$ is identified with a **one-hot vector** in $\mathbb{R}^M$.

For an input x_i belonging to class $\mathcal{C}_j$, the one-hot vector $y_i \in \mathbb{R}^M$ is a binary vector with a 1 in the jth position and zeros everywhere else:

$$y_i = [0, \; 0, \; \ldots, \; 0, \; \underset{\substack{\uparrow \\ j}}{1}, \; 0, \; \ldots, \; 0]$$

Multiple target dimensions: classification

In this case, the network model prediction needs to predict a categorical distribution over M classes. This can be achieved using a **softmax** output layer:

$$\begin{aligned} \hat{h}_L &= W^{(L-1)} h_{L-1} + b^{(L-1)}, \qquad \text{(pre-activations / logits)} \\ h_L^{(j)} &= \frac{e^{\hat{h}_L^{(j)}}}{\sum_{k=1}^{M} e^{\hat{h}_L^{(k)}}} \qquad \text{(softmax output)} \end{aligned}$$

In the above, $W^{(L-1)} \in \mathbb{R}^{M \times n_{L-1}}$, $b^{(L-1)} \in \mathbb{R}^M$ and $\hat{h}_L, h_L \in \mathbb{R}^M$.

Note that $h_L^{(j)} \geq 0$ and $\sum_{j=1}^{M} h_L^{(j)} = 1$.

The j-th coordinate of the output layer $h_L = f(x, \boldsymbol{\theta})$ gives the model probability $p(\mathcal{C}_j|x)$.

Multiple target dimensions: classification

As usual, we denote the model prediction $\hat{y}_i = f(x_i, \boldsymbol{\theta})$. The likelihood function in this case is given by

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}|\mathbf{x}, \mathbf{y}) &= \prod_{i=1}^{N} p(y_i|x_i, \boldsymbol{\theta}) \\ &= \prod_{i=1}^{N}\prod_{j=1}^{M} (\hat{y}_i^{(j)})^{y_i^{(j)}} \end{aligned}$$

The parameters that minimise the negative log-likelihood are given by:

$$\boldsymbol{\theta}_{ML} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N}\sum_{j=1}^{M} -y_i^{(j)} \log \hat{y}_i^{(j)}$$

and we have the following cross entropy loss function:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N}\sum_{j=1}^{M} -y_i^{(j)} \log \hat{y}_i^{(j)}$$

Outline

Backpropagation

Neural network training can be viewed iterating the following two separate stages:

1. Computation of the derivatives of the loss function with respect to the model parameters
2. Use the computed gradient to update the parameters

The first of these stages is done through applying the chain rule of differentiation. The process of computing these derivatives in an efficient manner is known as **backpropagation**.

In the remainder of this lecture, we will derive the backpropagation algorithm for finding the loss function derivatives for a general feedforward neural network architecture.

Backpropagation: preactivations a_j and activations z_j

- We view the network as a collection of neurons units
- Each unit has an *preactivation* value a_j, computed as

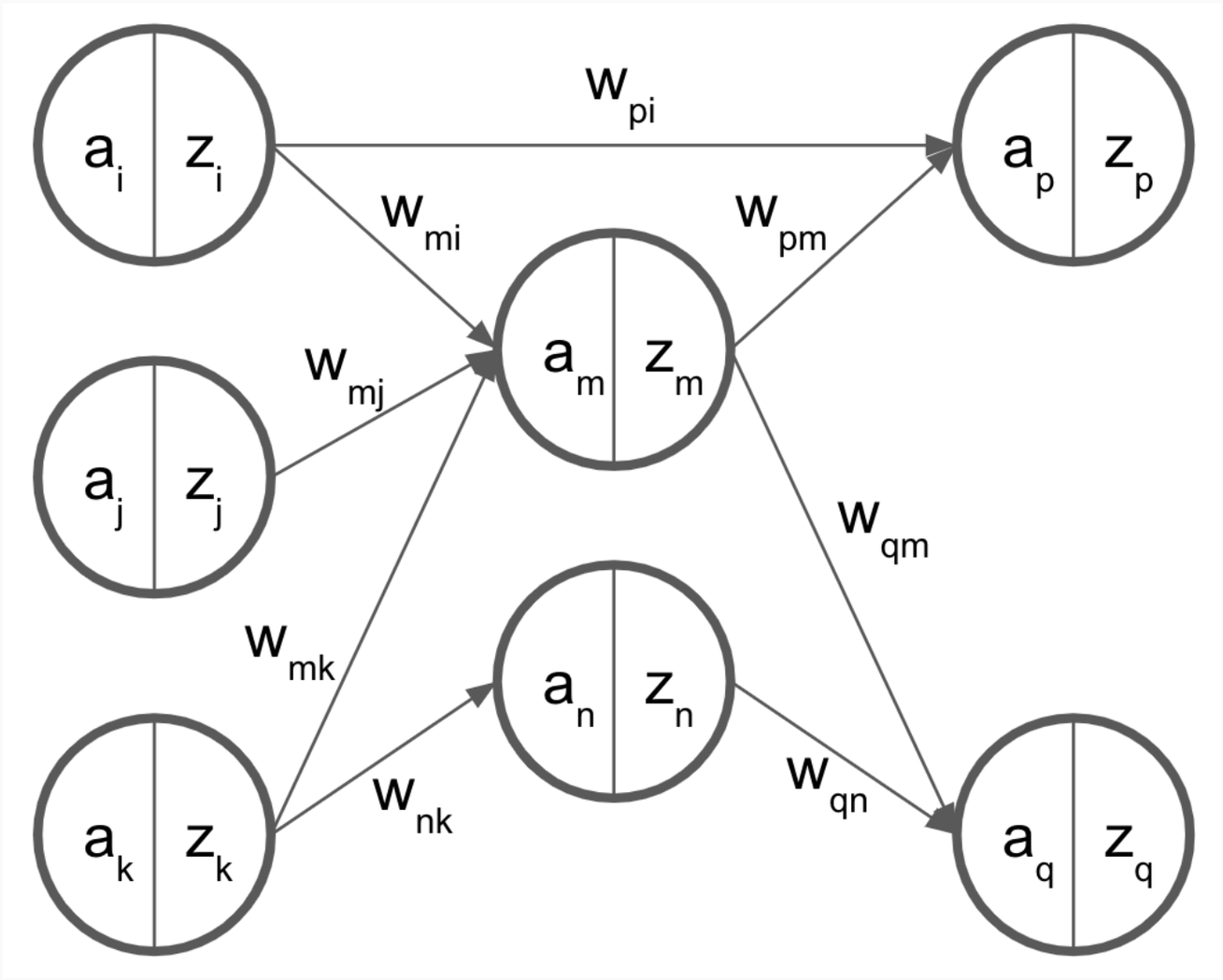
$$a_j = \sum_{k \in \uparrow j} w_{jk} z_k + b_j \tag{1}$$

  where $\uparrow j$ denotes the set of indices of neurons upstream from unit j, i.e. that feed directly into unit j
- Each neuron also has an *activation* value z_j given by

$$z_j = h(a_j) \tag{2}$$

  where $h(\cdot)$ is an activation function

Backpropagation: directed acyclic graph



Note the variables z_j could be used for input or output nodes to the network. The loss function can be seen as a function of one or more node activations and data labels.

Backpropagation: per-example loss function

- We note that due to the i.i.d. assumption, the loss is a sum of terms for each data example, e.g.

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(x_i, \boldsymbol{\theta})\right)^2$$

- therefore we will consider the loss derivatives computed on a single data example, in this example:

$$E_i(\boldsymbol{\theta}) = \left(y_i - f(x_i, \boldsymbol{\theta})\right)^2$$

- Recall that $\boldsymbol{\theta}$ denotes the network parameters
- We want to compute derivatives of $E_i(\boldsymbol{\theta})$ with respect to the weights w_{jk} and biases b_j of the network
- The derivatives of $E(\boldsymbol{\theta})$ are computed by averaging $E_i(\boldsymbol{\theta})$ over the minibatch

Error backpropagation

We first compute the *forward pass*, in which the input data is fed to the network and the preactivations a_j and activations z_j are computed in sequence corresponding to (1) and (2), for the current parameters $\boldsymbol{\theta}$.

Consider the derivative of E_i with respect to w_{jk} and b_j. We have:

$$\frac{\partial E_i}{\partial w_{jk}} = \frac{\partial E_i}{\partial a_j}\frac{\partial a_j}{\partial w_{jk}} = \frac{\partial E_i}{\partial a_j} z_k$$

and

$$\frac{\partial E_i}{\partial b_j} = \frac{\partial E_i}{\partial a_j}\frac{\partial a_j}{\partial b_j} = \frac{\partial E_i}{\partial a_j}$$

Error backpropagation

Introducing the notation $\delta_j := \frac{\partial E_i}{\partial a_j}$, called the **error**. We then write

$$\frac{\partial E_i}{\partial w_{jk}} = \delta_j z_k, \qquad \frac{\partial E_i}{\partial b_j} = \delta_j \tag{3}$$

We therefore need to compute the quantity δ_j for each hidden and output unit in the network. Again using the chain rule, we have

$$\begin{aligned} \delta_j \equiv \frac{\partial E_i}{\partial a_j} &= \sum_{k \in \downarrow j} \frac{\partial E_i}{\partial a_k} \frac{\partial a_k}{\partial a_j} \\ &= \sum_{k \in \downarrow j} \delta_k \frac{\partial a_k}{\partial a_j} \end{aligned}$$

where $\downarrow j$ denotes the set of indices of neurons downstream from unit j; that is, the neurons that unit j feeds into.

Error backpropagation

Combining equations (1) and (2) we see that

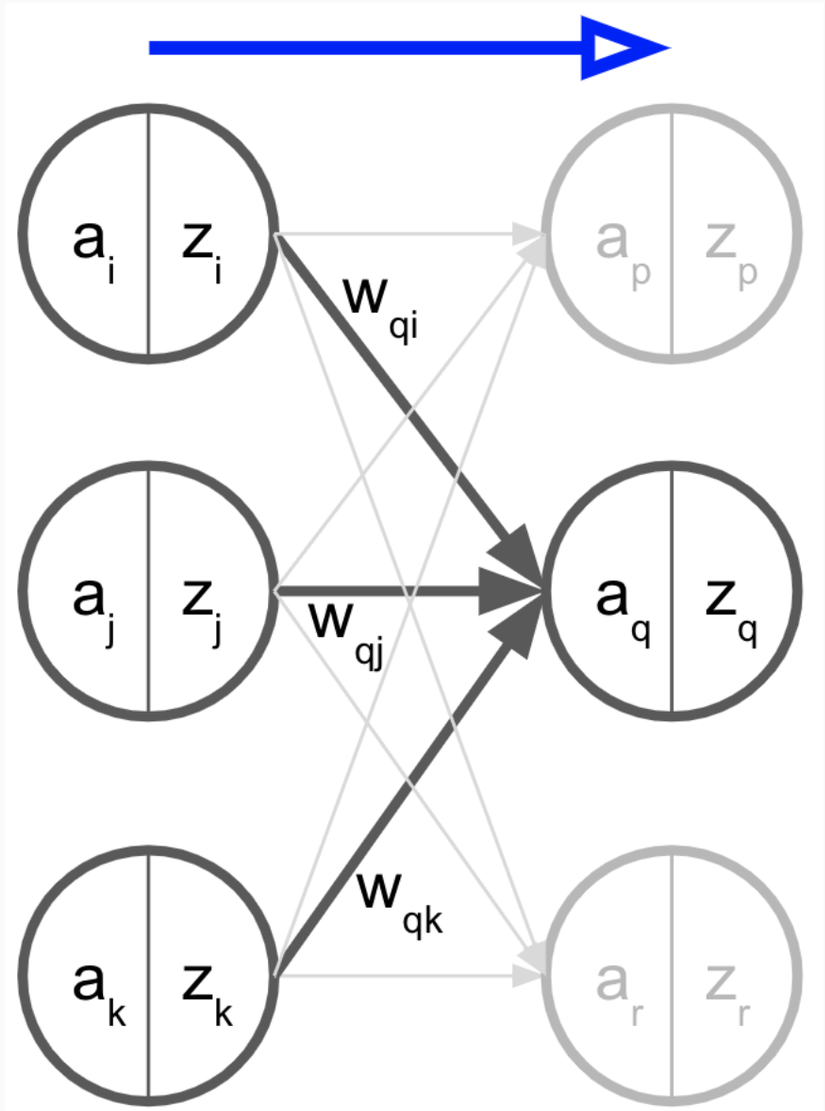
$$\begin{aligned} a_k &= \sum_{j \in \uparrow k} w_{kj} h(a_j) + b_k \\ \frac{\partial a_k}{\partial a_j} &= w_{kj} h'(a_j) \end{aligned}$$

where h' is the derivative of the activation function. So we have

$$\begin{aligned} \delta_j \equiv \frac{\partial E_i}{\partial a_j} &= \sum_{k \in \downarrow j} \delta_k \frac{\partial a_k}{\partial a_j} \\ &= h'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k \end{aligned}$$

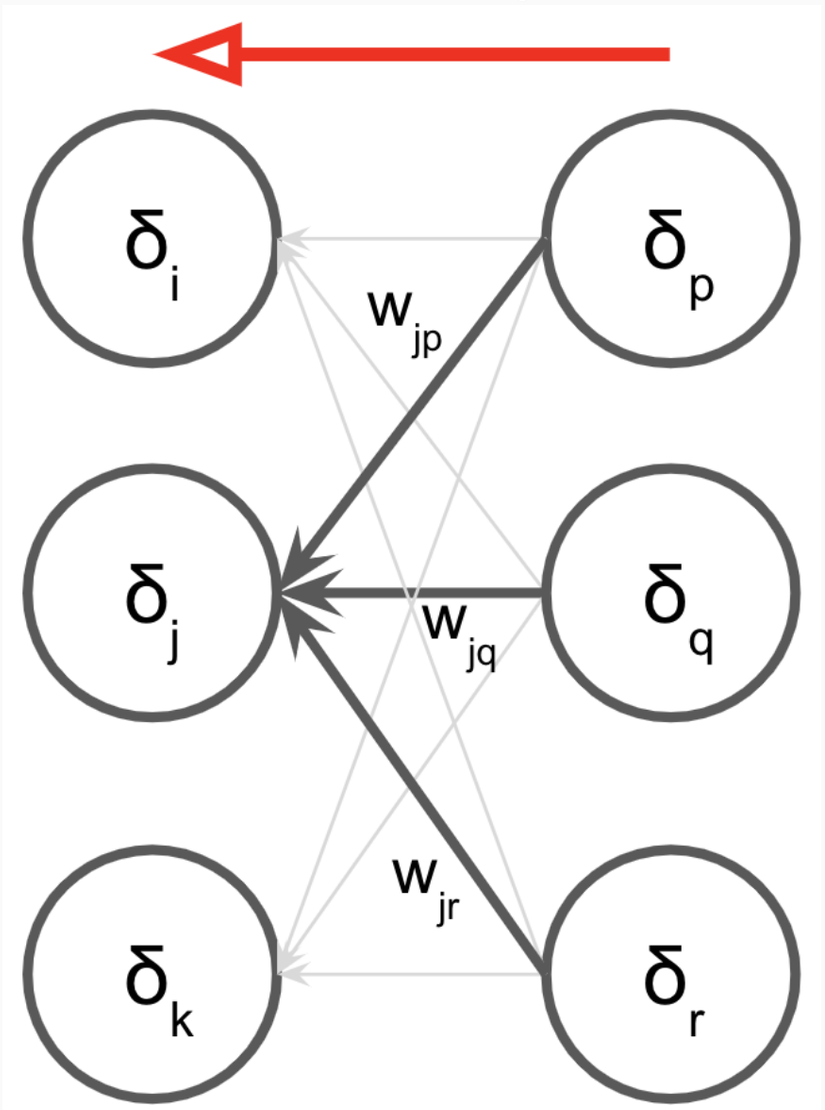
Forward and backward passes

Forward pass



$$a_q = \sum_{j \in \uparrow k} w_{qj} h(a_j) + b_q$$

Backward pass



$$\delta_j = h'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k$$

Backpropagation algorithm

$$\text{Forward pass:} \quad a_k = \sum_{j \in \uparrow k} w_{kj} h(a_j) + b_k$$

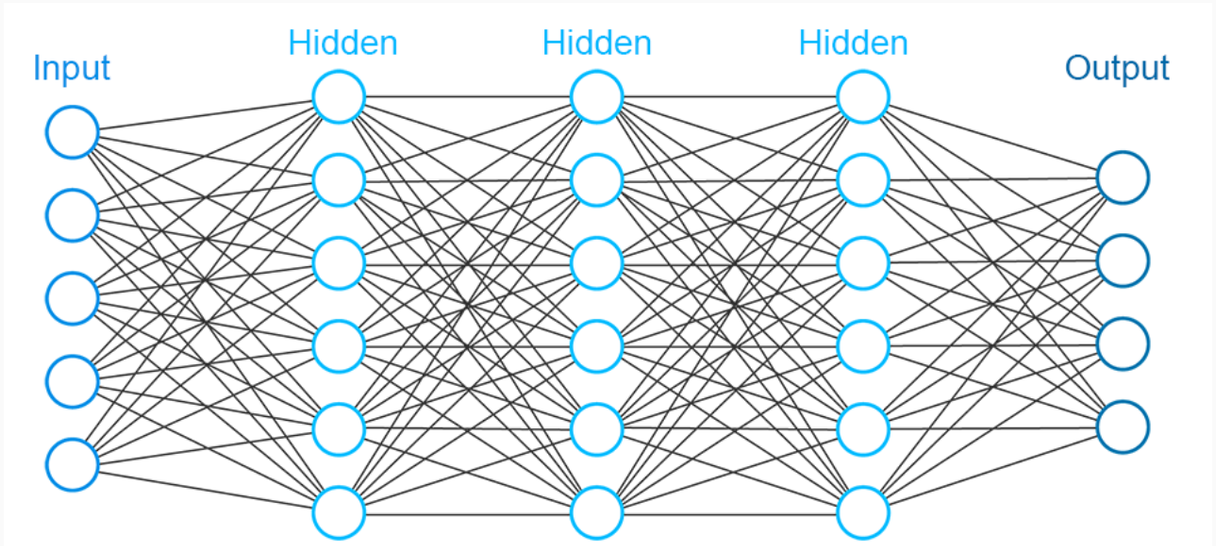
$$\text{Backward pass:} \quad \delta_j = h'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k$$

$$\text{Derivatives:} \quad \frac{\partial E_i}{\partial w_{jk}} = \delta_j z_k, \qquad \frac{\partial E_i}{\partial b_j} = \delta_j$$

1. First, propagate the signal forwards by passing an input vector x_i to the network and computing all pre-activations and activations
2. Evaluate $\delta_k = \frac{\partial E_i}{\partial a_k}$ for the output neuron pre-activations
3. Backpropagate the errors δ to compute δ_j for each hidden unit
4. Compute the derivatives using the errors δ_j

Feedforward network / MLP

We can rewrite the backpropagation equations in matrix-vector form for the feedforward network / MLP architecture:



Suppose our network has L layers. Denote the j-th pre/post-activation in the k-th layer as $a_j^{(k)}$ and $z_j^{(k)}$ respectively, and $a^{(k)}, z^{(k)}, \delta^{(k)} \in \mathbb{R}^{n_k}$ are the collections of pre-activations/activations/errors in the k-th layer.

The input layer is $z^{(1)} \equiv x_i$ and output layer is $z^{(L)} \equiv \hat{y}_i$.

Feedforward network / MLP

Denote the weight matrices and biases that map from layer k to $k+1$ as $W^{(k)}$ and $b^{(k)}$ respectively.

The forward and backward pass equations become:

$$\begin{aligned} \text{Forward pass:} \quad a^{(k+1)} &= W^{(k)} z^{(k)} + b^{(k)} \\ z^{(k)} &= h(a^{(k)}) \end{aligned}$$

$$\text{Backward pass:} \quad \delta^{(k)} = H'(a^{(k)}) \left(W^{(k)}\right)^T \delta^{(k+1)}$$

where $H'(a^{(k)}) = \text{diag}[h'(a^{(k)})]$, and the derivatives are given by

$$\begin{aligned} \frac{\partial L}{\partial W^{(k)}} &= \delta^{(k+1)} \left(z^{(k)}\right)^T \\ \frac{\partial L}{\partial b^{(k)}} &= \delta^{(k+1)} \end{aligned}$$

Feedforward network / MLP

In the final layer, the error is

$$\delta^{(L)} = H'(a^{(L)}) \frac{\partial E_i}{\partial z^{(L)}}(\hat{y}_i)$$

and so we find that

$$\delta^{(k)} = \left(\prod_{j=k}^{L-1} H'(a^{(j)}) (W^{(j)})^T \right) H'(a^{(L)}) \frac{\partial E_i}{\partial z^{(L)}}(\hat{y}_i) \tag{4}$$

Outline

Vanishing and exploding gradients

Equation (4) highlights one of the main issues with training neural networks; that of **vanishing** or **exploding** gradients.

- Some activation functions (such as sigmoid) can saturate, leading to small values in the diagonal matrix $H'(a^{(j)})$
- This can lead to a slow down in the learning of weights
- Deep networks can suffer from vanishing or exploding gradients if weight matrices have large or small eigenvalues
- This problem can be tackled through network design and initialisation strategies

Xavier initialisation

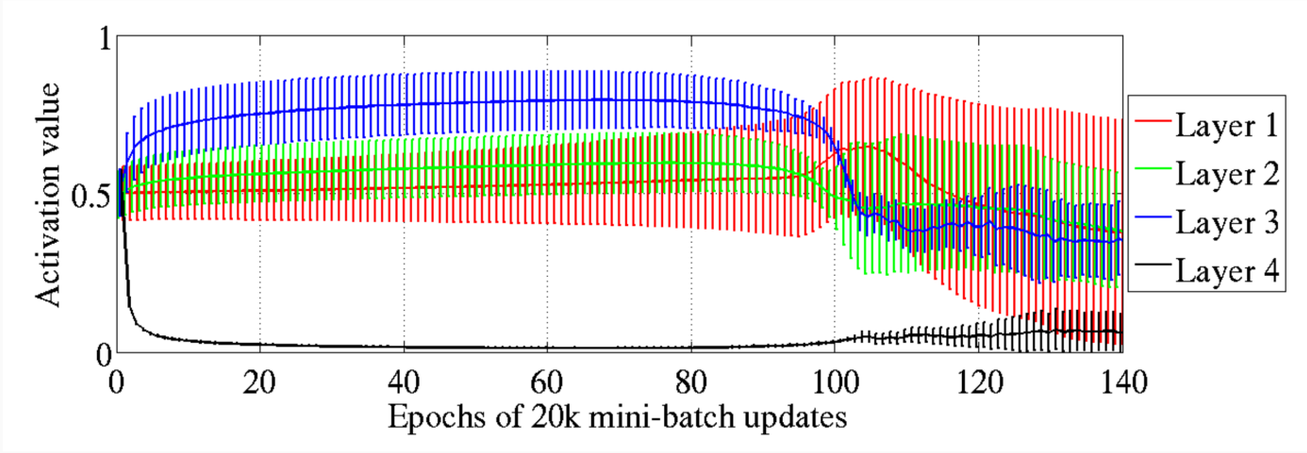
Initialisation of the weights of a neural network is important to ensure the forward and backward signals can propagate through the network during training.

One method that has become popular is commonly referred to as **Xavier** or **Glorot initialisation**.

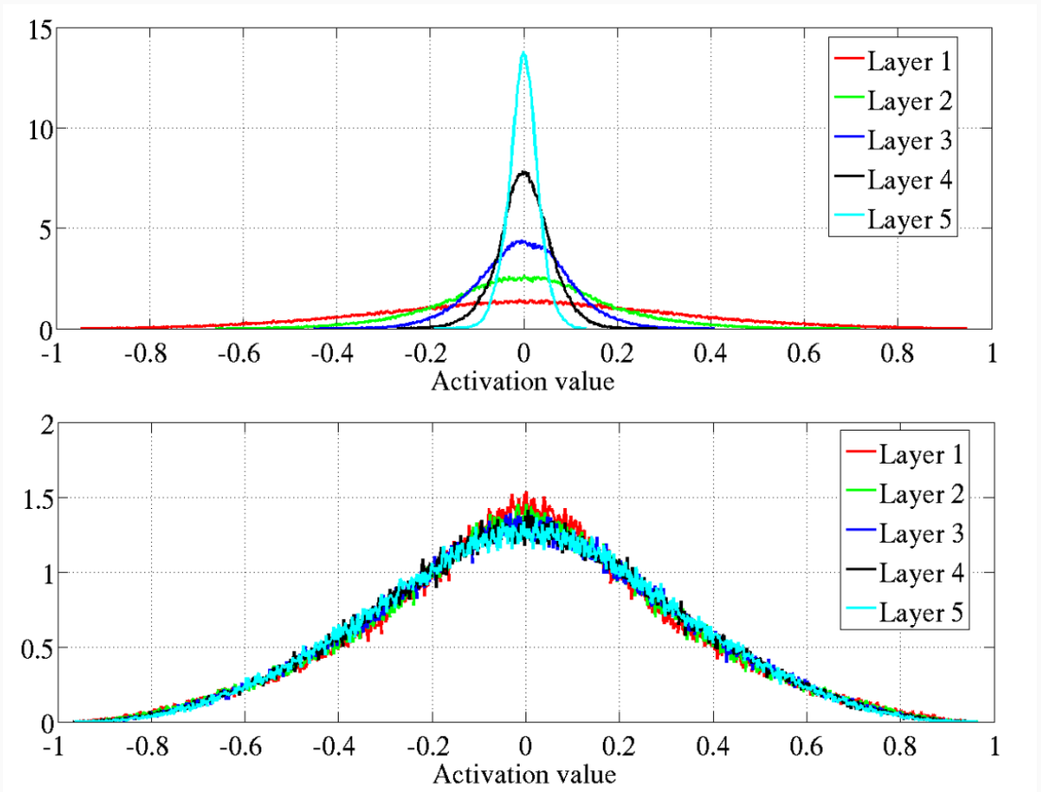
The aim of this initialisation is to specify basic statistical properties of the weight distribution to ensure signal propagation.

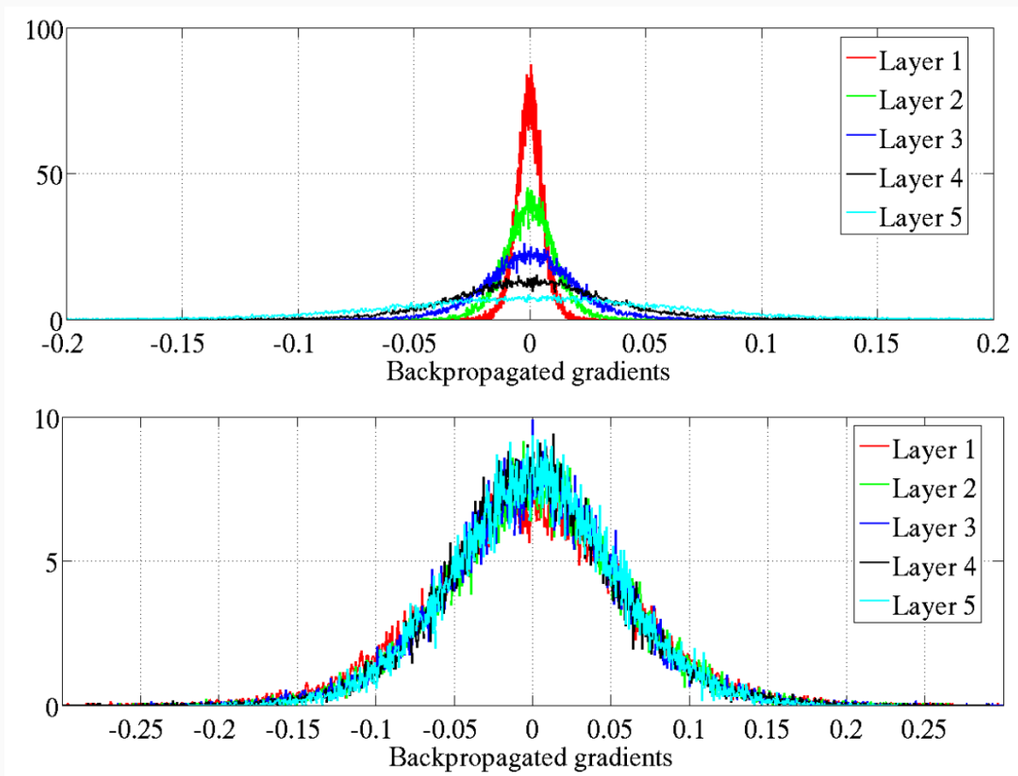
- Fix a zero mean for the weight distribution
- We want to find the optimal variance of the weights in each layer

Xavier initialisation



Mean and standard deviation of activation values. Note the quick saturation of the top layer.





Activation & gradient histograms. Top: Unnormalised initialisation. Bottom: Xavier initialisation.

source: X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

Xavier initialisation

Assume:

- The activations are in the linear region, $H'(a_j) \approx I$.
- The network input features $x_i^{(j)}$ are zero mean and i.i.d.
- The weight distribution is i.i.d with zero mean for each layer

Then the variance for each unit in the k-th layer $z^{(k)}$ is given by

$$\mathrm{Var}[z^{(k)}] = \mathrm{Var}[x_i] \prod_{k'=1}^{k-1} n_{k'} \mathrm{Var}[W^{(k')}],$$

where $\mathrm{Var}[x_i]$ and $\mathrm{Var}[W^{(j)}]$ denote the shared variances throughout the input layer $z^{(1)} = x_i$ and the weight matrix $W^{(j)}$ respectively.

Xavier initialisation

Also we can show that

$$\mathrm{Var}[\delta^{(k)}] = \mathrm{Var}[\nabla_{h_L=y_i} E_i]] \prod_{k'=k}^{L-1} n_{k'+1} \mathrm{Var}[W^{(k')}],$$

where recall $\delta^{(k)} := \frac{\partial E_i}{\partial a^{(k)}}$.

To preserve the signal through the network in both the forward and backward passes, we want

$$\begin{aligned} \mathrm{Var}[z^{(k)}] \approx \mathrm{Var}[z^{(k')}] \quad &\Rightarrow \quad n_k \mathrm{Var}[W^{(k)}] = 1 \quad \forall k \\ \mathrm{Var}[\delta^{(k)}] \approx \mathrm{Var}[\delta^{(k')}] \quad &\Rightarrow \quad n_{k+1} \mathrm{Var}[W^{(k)}] = 1 \quad \forall k \end{aligned}$$

Xavier initialisation

As a compromise between these two constraints, the suggested **Xavier initialisation** scheme is given by

$$\mathrm{Var}[W^{(k)}] = \frac{2}{n_k + n_{k+1}} \quad \forall k = 1, \ldots, L-1.$$

An example weight distribution to use for initialisation with this scheme is:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_k + n_{k+1}}}, \frac{\sqrt{6}}{\sqrt{n_k + n_{k+1}}}\right]$$

Outline

Neural network optimisers

Recall the two stages of training neural networks:

1. Computation of the derivatives of the loss function with respect to the model parameters (backpropagation)
2. Use the computed gradient to update the parameters

We have looked at the first stage and the backpropagation algorithm.

We have also seen how vanishing or exploding gradients can occur, and how we can tackle them with good initialisation strategies.

We now turn to the second stage of neural network training, and examine some popular optimisation algorithms.

Gradient descent

Batch gradient descent is the most straightforward variant of gradient descent, where the gradient of the loss function is taken over the entire training set and the parameters adjusted accordingly:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}),$$

where η is the learning rate.

- Very slow, due to the use of the whole dataset
- Intractable for large datasets
- Inefficient use of the data

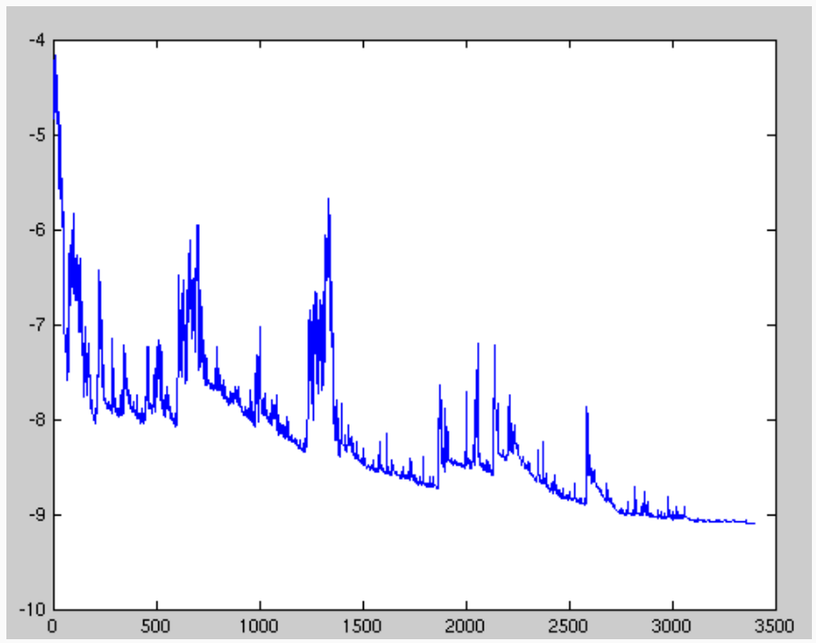
Stochastic gradient descent

Stochastic gradient descent divides the dataset into smaller batches, and computes the gradient for each update on this smaller batch:

$$\begin{aligned} \mathbf{g} &= \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} E_i(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}), \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \mathbf{g}, \end{aligned}$$

where again η is the learning rate.

- Reduces redundancy in gradient computation
- Faster (and usually better) convergence
- Can be used online



Some challenges

- Convergence with SGD can be very slow
- Setting the learning rate can be difficult, and often involves trial and error
- Learning rate schedules can be used to reduce the learning rate over the course of training
  - Different weights might operate on different scales, and require different rates of learning
- Local minima, and/or saddle points

Several optimisation algorithms have been proposed to help treat these problems.

Momentum

One common tweak to accelerate the slow convergence of the SGD algorithm is to add momentum:

$$\begin{aligned}
\mathbf{g}^k &= \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} E_i(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}^k) \\
\boldsymbol{z}^{k+1} &= \beta \boldsymbol{z}^k + \mathbf{g}^k \\
\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \eta \boldsymbol{z}^{k+1}
\end{aligned}$$

Setting $\beta = 0$ recovers SGD.

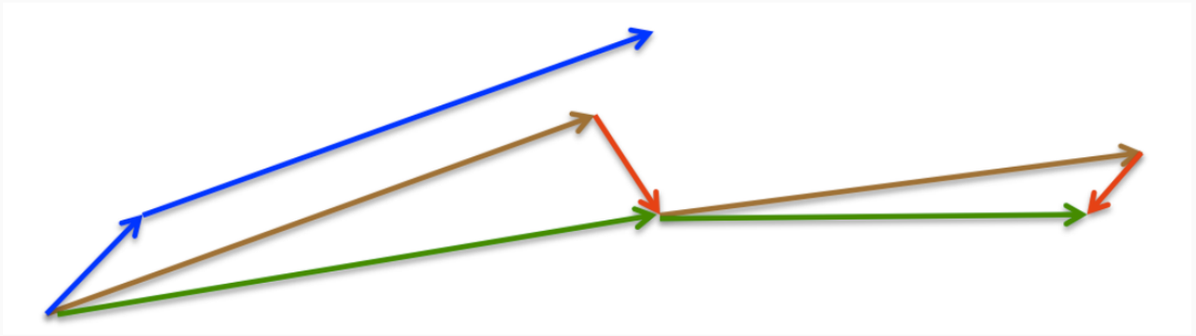
In practice, typical β values when using momentum is around 0.9.

Nesterov momentum

A common variant of momentum is to use Nesterov momentum, which computes the gradient correction after the accumulated gradient, instead of before:

$$\left.\begin{array}{rcl} \mathbf{z}^{k+1} & = & \beta \mathbf{z}^k + \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k) \\ \boldsymbol{\theta}^{k+1} & = & \boldsymbol{\theta}^k - \eta \mathbf{z}^{k+1} \end{array}\right\} \quad \text{(Momentum update)}$$

$$\left.\begin{array}{rcl} \mathbf{z}^{k+1} & = & \beta \mathbf{z}^k + \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k - \eta\beta \mathbf{z}^k) \\ \boldsymbol{\theta}^{k+1} & = & \boldsymbol{\theta}^k - \eta \mathbf{z}^{k+1} \end{array}\right\} \quad \text{(Nesterov momentum)}$$



blue: SGD with momentum

green: SGD with Nesterov momentum

source: G. Hinton's Coursera lectures

Adagrad

- Adapts the learning rate for each parameter
- Less frequent (active) parameters receive larger updates
- Well suited to sparse data
- Used to train GloVe word embeddings

The update rule is (division and square root performed element-wise):

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \frac{\eta}{\sqrt{G^{k} + \epsilon}} \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^{k}),$$

where $G^k \in \mathbb{R}^{p \times p}$ is a diagonal matrix where the diagonal elements G^k_{ii} are the sum of squares of gradients with respect to θ_i up to time step k.

Note that the resulting learning rates per parameter are monotonically decreasing, and eventually the algorithm effectively stops learning.

RMSProp

- Aims to resolve the vanishing learning rates of Adagrad
- Unpublished method, appears in G. Hinton's Coursera class
- Uses a decaying average of past squared gradients

The update rule is:

$$\begin{aligned}
\mathbb{E}[\mathbf{g}^2]^k &= \gamma \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\gamma)(\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k))^2 \\
\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \frac{\eta}{\sqrt{\mathbb{E}[\mathbf{g}^2]^k + \epsilon}} \odot \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k),
\end{aligned}$$

As before, the division and square root are performed element-wise, and $\odot$ is the Hadamard product. The γ term is typically set similar to momentum, around 0.9.

Adadelta

- Independently developed around the same time as RMSProp
- Also aims to resolve the vanishing learning rates of Adagrad
- Removes the need to set a default learning rate

The update rule is:

$$
\begin{aligned}
\mathbb{E}[\mathbf{g}^2]^k &= \gamma \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\gamma)(\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k))^2 \\
\mathbb{E}[(\Delta\boldsymbol{\theta})^2]^k &= \gamma \mathbb{E}[(\Delta\boldsymbol{\theta})^2]^{k-1} + (1-\gamma)(\boldsymbol{\theta}^k - \boldsymbol{\theta}^{k-1})^2 \\
\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \frac{\sqrt{\mathbb{E}[(\Delta\boldsymbol{\theta})^2]^k + \epsilon}}{\sqrt{\mathbb{E}[\mathbf{g}^2]^k + \epsilon}} \odot \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^k),
\end{aligned}
$$

The γ term is typically set similar to momentum, around 0.9.

Adam

- Ada<u></u>ptive <u>m</u>oment estimation
- Also computes adaptive learning rates per parameter
- Estimates first and second moments of the gradients

The update rule is:

$$\begin{aligned}
\mathbb{E}[\mathbf{g}]^k &= \beta_1 \mathbb{E}[\mathbf{g}]^{k-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k), & \mathbf{m}^k &= \mathbb{E}[\mathbf{g}]^k/(1-\beta_1) \\
\mathbb{E}[\mathbf{g}^2]^k &= \beta_2 \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\beta_2)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k))^2, & \mathbf{v}^k &= \mathbb{E}[\mathbf{g}^2]^k/(1-\beta_2)
\end{aligned}$$

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\eta}{\sqrt{\mathbf{v}^k + \epsilon}} \odot \mathbf{m}^k$$

- $\mathbf{m}^k$ and $\mathbf{v}^k$ correct for an initial bias towards zero
- Typical values are $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$ and $\epsilon \approx 10^{-8}$.

Optimisers summary

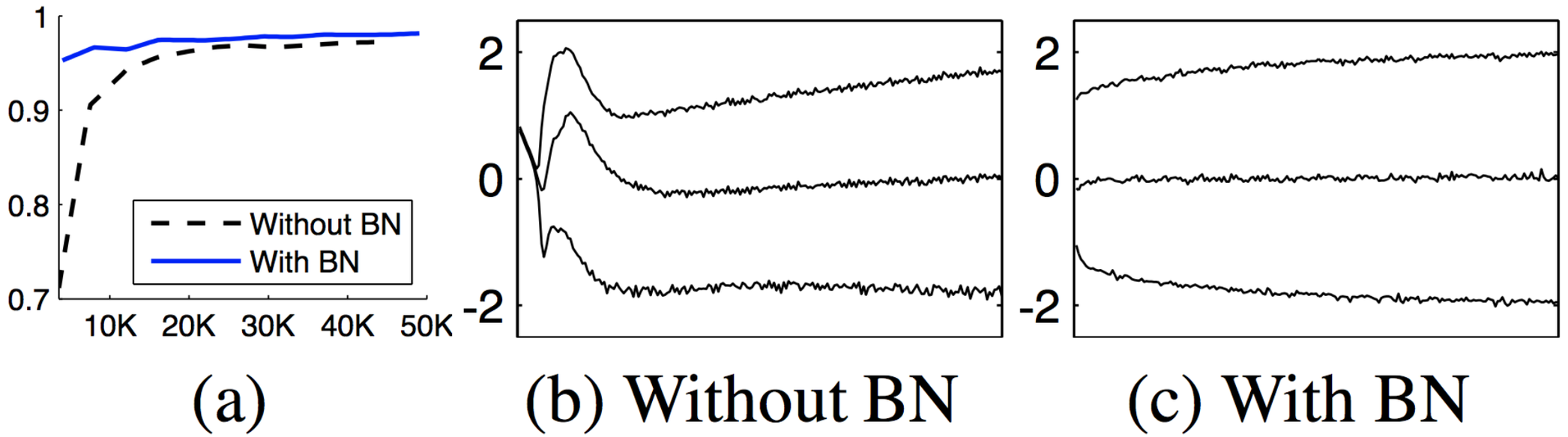
- Adagrad introduces adaptive learning rate; best suited for sparse data
- RMSProp resolves the vanishing learning rates by using decaying averages
- Adadelta is similar to RMSProp but does not require a learning rate
- Adam adds bias-correction and momentum
- SGD is still often used and tends to find a good minimiser and generalise well
- Further work on optimisers includes warm restarts, switching and cyclic learning rates

Outline

Batch normalization

- Widely used method in training deep neural networks
- Normalises the distribution of activations throughout the network
- Original paper defines *internal covariate shift* as the change in distribution of a network layer's activations as parameters are changed.
- Ideally we would like to whiten each layer's activations, but this is too expensive
    - Instead, normalize each feature independently in each layer
    - Use minibatch statistics to approximate the statistics of the training set

Batch normalization



(a) Test accuracy on a network trained on MNIST against training iterations

(b, c) Evolution of typical distributions to a sigmoid layer, shown as {15, 50, 85}th percentiles

source: S. Ioffe and C. Szegedy, "Batch normalisation: accelerating deep network training by reducing internal covariate shift", (2015).

Batch normalization

Note that we must take account of any normalization in the gradient computation:

- Suppose $x = u + b$ for some input u
- Normalize $\hat{x} = x - \mathbb{E}[x]$, with the expectation taken over the training set
- Compute update $\Delta b \propto -\partial L / \partial \hat{x}$
- $b \leftarrow b + \Delta b$
- Then $u + (b + \Delta b) - \mathbb{E}[u + (b + \Delta b)] = u + b - \mathbb{E}[u + b]$
- $\Rightarrow$ Layer output doesn't change
- Normalization needs to be part of the network structure, and activations always follow the desired distribution

Batch normalization

For a layer with d-dimensional input $\mathbf{x} = (x^{(1)}, \ldots, x^{(d)})$, normalize each dimension

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

In order to maintain full expressive power of the network, we make sure the final transformation can represent the identity:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Here, $\gamma^{(k)}$ and $\beta^{(k)}$ are additional learned parameters. Note that setting $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$ recovers the original activations.

Batch normalization

Statistics $\mathbb{E}[x^{(k)}]$ and $\mathrm{Var}[x^{(k)}]$ are computed over each minibatch $\mathcal{B} = \{x_1, \ldots, x_m\}$.

$$
\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma^2_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \\
\hat{x}_i &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \\
y_i &= \gamma\hat{x}_i + \beta =: BN_{\gamma,\beta}(x_i)
\end{aligned}
$$

Population statistics are also computed and are used for inference in place of the minibatch statistics.

Batch normalization

For backpropagation, the normalization is taken into account:

$$\begin{aligned}
\frac{\partial L}{\partial \hat{x}_i} &= \frac{\partial L}{\partial y_i} \cdot \gamma \\
\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \sigma_{\mathcal{B}}^2} \\
\frac{\partial L}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu_{\mathcal{B}}} \right) + \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mu_{\mathcal{B}}} \\
\frac{\partial L}{\partial x_i} &= \left(\frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial x_i} \right) + \left(\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\partial \sigma_{\mathcal{B}}^2}{\partial x_i} \right) + \left(\frac{\partial L}{\partial \mu_{\mathcal{B}}} \cdot \frac{\partial \mu_{\mathcal{B}}}{\partial x_i} \right) \\
\frac{\partial L}{\partial \gamma} &= \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \hat{x}_i \\
\frac{\partial L}{\partial \beta} &= \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}
\end{aligned}$$